

```
trace("Pratique  
d'ActionScript  
3");
```

```
// Thibault IMBERT
```

# Remerciements

//

"Je voudrais remercier tout particulièrement Mathieu Anthoine (Mama) pour sa relecture soignée et ses conseils avisés durant toute la rédaction de l'ouvrage. Son expérience et sa vision du Flash est une source d'inspiration. Jérôme Decoster pour sa relecture attentive, son soutien et ses conseils toujours pertinents. Pour nos soirées pizza à discuter ActionScript 3 et te faire rentrer chez toi en vélib.

Chris Georgenes ([www.mudbubble.com](http://www.mudbubble.com)) et Aurelien Marcheguy de Carton Blanc ([www.cartonblanc.com](http://www.cartonblanc.com)) et Stéphane Mortka pour leur soutien et leurs illustrations qui m'ont permis d'enrichir graphiquement l'ouvrage.

Je remercie tous mes compagnons flasheurs qui m'ont forcément soutenu et aidé à un moment donné : Joa Ebert, Didier Brun, Jérôme Cordiez, Vincent Maitray, David Deraedt, Frédéric Saunier, Benjamin Jung.

Le centre Regart.net pour m'avoir soutenu. A Nicolas pour ses opinions, à Eric pour ses bonnes blagues, et à Guylaine pour ces années chez Regart.

Justin Everett-Church d'Adobe et Henri Torgemane de Yahoo pour leur aide et leur soutien.

Ségolène ([www.chevaldetroie.net](http://www.chevaldetroie.net)) pour avoir relu tous ces chapitres à la recherche de fautes d'orthographe.

Mes amis que je n'ai pas vus pendant quasiment 8 mois : Sébastien, Bessam, Laurent, Paul, Juan, Pascal, Bob Groove et bien sûr Stevie Wonder.

**Sonia** pour ce dimanche 20 août 2006 où ma vie a changée.

Mes parents et à ma sœur qui m'ont soutenu pendant l'écriture.

Bien entendu, je ne peux pas terminer les remerciements sans remercier tous mes stagiaires que j'ai pu rencontrer depuis plus de trois ans de formations. J'ai appris énormément chaque jour à vos côtés, vous avez participé à cet ouvrage d'une façon ou d'une autre."

# Préface

//

"Je pratique Flash depuis maintenant 10 ans et j'ai pu suivre de près toutes ses évolutions. Flash 3 a accompagné mes premiers pas dans l'univers du web. J'ai découvert un logiciel incroyablement accessible et dont l'ergonomie astucieuse, les outils d'animation et d'interactivité répondaient parfaitement aux besoins de l'époque.

Quelques années plus tard, un bond de géant est franchi avec Flash MX et l'apparition de la première version robuste du langage ActionScript. Flash est alors un outil mûr et équilibré. Au même moment, sortent les premières versions de Flash Media Server et Flash Remoting qui deviennent des compléments indispensables à tout développement ambitieux.

Suivant l'évolution des créations web de plus en plus dynamiques et complexes, Flash bascule, non sans mal, vers un outil de développement plus sophistiqué (Flash MX 2004), nécessitant la mise en place d'équipes spécialisées. L'époque du flasheur maîtrisant tout de A à Z est révolue. Flash s'adresse désormais à deux publics distincts : graphistes d'un côté, développeurs de l'autre.

L'équilibre devient alors précaire et la tendance à favoriser le code au détriment de l'accessibilité de l'outil se développe. Les réalisations du graphiste ou de l'animateur sont maintenant souvent subordonnées au travail du développeur, ce qui ne favorise pas toujours la création.

Si Flash 8, grâce aux filtres et mélanges, ouvre de nouvelles voies à la création et redonne un peu de place au graphiste, c'est néanmoins, le plus souvent, le développeur qui contrôle le projet.

Flash 9 et l'ActionScript 3 représentent une nouvelle étape vers la technicisation de l'outil. Connaître « l'environnement auteur » ne suffit plus, il est indispensable de comprendre et de maîtriser les rouages du lecteur pour éviter la réalisation d'applications totalement instables.

Etant au cœur de la production, j'ai toujours déploré que les ouvrages Actionscript prennent si peu en compte les réalités que peuvent rencontrer une agence, un studio, une équipe ou tout professionnel dans son travail quotidien.

J'apprécie particulièrement l'ouvrage de Thibault pour les exemples concrets, directement utilisables en production, qu'il nous livre. Il nous dispense justement d'explications trop virtuelles ou d'un esthétisme superflu du code.

Fort de son expérience de formateur, il reste simple et clair au moyen d'un vocabulaire précis, sans jamais tomber, ni dans le perfectionnisme lexical, ni dans la vulgarisation imprécise.

Cet ouvrage va bien au delà de l'apprentissage du langage ActionScript 3 en posant les bases de sa bonne utilisation dans Flash et en fournissant un éclairage neuf et pertinent sur cet outil fabuleux."

**Mathieu Anthoine**

Game Designer et co-fondateur du studio Yamago [www.yamago.net](http://www.yamago.net)

```
trace("Pratique d'ActionScript 3");
```

```
// Pratique d'ActionScript 3 s'adresse à tous les flasheurs.
```

Cet ouvrage dresse un panorama de l'utilisation d'ActionScript 3 et de ses nouveautés, ainsi que du nouveau lecteur Flash 9.

L'auteur explique au travers de nombreux exemples et cas concrets comment traiter désormais les objets, le texte, le son et la vidéo, le chargement et l'envoi de données externes (variables, XML, etc.), les sockets, etc.

Certains sujets avancés et peu abordés comme les classes bas niveau ou encore Flash Remoting sont également traités.

Enfin, l'auteur livre une application complète qui reprend les différents points évoqués.

```
// Thibault IMBERT est aujourd'hui ingénieur système chez Adobe France.
```

Il a commencé en tant que développeur Flash pour différentes agences, avant de rejoindre Regart.net comme formateur et responsable pédagogique pour toutes les formations concernant la plate-forme Flash.

Pendant son temps libre, Thibault expérimente ActionScript 3, comme en témoigne son site [www.bytearray.org](http://www.bytearray.org). Il travaille également sur différents projets tels que WiiFlash [www.wiiflash.org](http://www.wiiflash.org) ou AlivePDF [www.alivepdf.org](http://www.alivepdf.org)





# 1

## Qu’est ce que l’ActionScript 3

<b>HISTORIQUE.....</b>	<b>1</b>
<b>10 RAISONS DE CODER EN ACTIONSCRIPT 3.....</b>	<b>3</b>
<b>OUTILS.....</b>	<b>4</b>
<b>LA PLATEFORME FLASH.....</b>	<b>4</b>

### Historique

Expliquer comment créer un projet AS dans Flex Builder

C’est en 1996 que l’aventure Flash commence lorsque la firme Macromedia rachète la petite société *FutureWave* auteur d’un logiciel d’animation vectoriel nommé *Future Splash Animator*.



*Figure 1-1. Future Splash Animator sorti en avril 1996.*

Développé à l'origine pour pouvoir animer du contenu vectoriel sur Internet, *Future Splash Animator* intégrait déjà les bases de Flash en matière d'animation mais ne possédait à cette époque aucun langage de programmation associé. Macromedia renomma alors le logiciel sous le nom de Flash 1.0 en 1996, mais il faut attendre 1999 afin que la notion de programmation fasse officiellement son apparition avec Flash 4.

Les documentations de l'époque ne parlent pas encore de langage ActionScript mais plus simplement d'actions. Grâce à celles-ci, il devient possible d'ajouter des comportements avancés aux boutons et autres objets graphiques. De nombreux graphistes à l'aise avec la programmation commencèrent à développer des comportements avancés et se mirent à échanger des scripts par le biais de forums et autres plateformes communautaires. Flash connu alors un engouement fulgurant et devint rapidement un outil d'animation avancé, capable de produire différents types de contenus interactifs comme des sites internet, des jeux, ou des applications multimédia.

C'est en 2001 que le langage ActionScript fait officiellement apparition au sein de Flash 5. La notion de syntaxe pointée est intégrée au langage qui suit pour la première fois les spécifications ECMAScript. Nous reviendrons sur cette notion au cours du prochain chapitre intitulé *Langage et API*.

Afin de répondre à une demande de la communauté pour un langage ActionScript plus structuré, Macromedia développe alors une nouvelle version d'ActionScript et l'intègre en 2003 au sein de Flash MX 2004 sous le nom d'ActionScript 2.0. Cette nouvelle version répond aux besoins des développeurs en offrant un langage orienté objet et non procédural comme c'était le cas en ActionScript 1.0.

A l'époque, Macromedia fait le choix d'une migration en douceur, et propose un langage ActionScript 2.0 souple permettant aux développeurs et graphistes de coder au sein d'un même projet en ActionScript 1.0 et 2.0. Si cette nouvelle version du langage ne satisfait pas les développeurs puristes, elle permet néanmoins aux développeurs débutant de migrer doucement vers un développement orienté objet.

Macromedia développe alors une nouvelle version 3.0 du langage ActionScript afin d'offrir des performances optimales à leur nouvelle création nommée Flex. Deux ans plus tard, la société Macromedia se voit rachetée par le géant Adobe et l'ActionScript 3 voit le jour au sein de Flash en 2007, lors de la sortie de Flash CS3.

## 10 raisons de coder en ActionScript 3

Si vous n'êtes pas encore convaincu de migrer vers ActionScript 3, voici 10 raisons pour ne plus hésiter :

- En décembre 2007, le lecteur Flash 9 possède un taux de pénétration de plus de 95%. Il demeure le lecteur multimédia le plus présent sur Internet.
- La vitesse d'exécution du code est environ 10 fois supérieure aux précédentes versions d'ActionScript.
- ActionScript 3 offre des possibilités incomparables aux précédentes versions d'ActionScript.
- Le code ActionScript 3 s'avère plus logique que les précédentes versions du langage.
- Le langage ActionScript 3 permet de développer du contenu en Flash, Flex, ou AIR. Nous reviendrons très vite sur ces différents frameworks.
- Il n'est pas nécessaire de connaître un autre langage orienté objet au préalable.
- ActionScript 3 est un langage orienté objet. Sa connaissance vous permettra d'aborder plus facilement d'autres langages objets tels Java, C#, ou C++.
- Le langage ActionScript 3 et l'API du lecteur Flash ont été entièrement repensés. Les développeurs ActionScript 1 et 2 seront ravis de découvrir les nouveautés du langage et la nouvelle organisation de l'API du lecteur.

- La technologie Flash ne cesse d'évoluer, vous ne risquez pas de vous ennuyer.
- ActionScript 3 est un langage souple, qui demeure ludique et accessible.

Bien entendu, vous pouvez mémoriser ces différents points afin de convaincre vos amis au cours d'une soirée *geek*.

ActionScript 3 ne vous sera pas d'une grande utilité sans un environnement de développement dédié. Nous allons nous attarder quelques instants sur les différents outils disponibles permettant de produire du contenu ActionScript 3.

## Outils

Afin qu'il n'y ait pas de confusions, voici les différents outils vous permettant de coder en ActionScript 3 :

- Flash CS3 : permet le développement d'animations et d'applications en ActionScript 3. Pour plus d'informations :  
<http://www.adobe.com/fr/products/flash/>
- Flex Builder : il s'agit d'un environnement auteur permettant le développement d'applications riches (RIA). Flex repose sur deux langages, le MXML afin de décrire les interfaces, et ActionScript pour la logique. Pour plus d'informations :  
<http://www.adobe.com/fr/products/flex/>
- Eclipse (SDK Flex et AIR) : les SDK de Flex et AIR permettent de produire du contenu Flex et AIR gratuitement. Il s'agit de plugins pour l'environnement de développement Eclipse.

Vous avez dit plateforme ?

## La plateforme Flash

Par le terme de plateforme Flash nous entendons l'ensemble des technologies à travers lesquelles nous retrouvons le lecteur Flash.

Nous pouvons diviser la plateforme Flash en trois catégories :

- Les lecteurs Flash : parmi les différents lecteurs, nous comptons le lecteur Flash, le lecteur Flash intégré à AIR, ainsi que Flash Lite (notons que Flash lite n'est pas à ce jour compatible avec ActionScript 3).
- Les outils de développement : Flex Builder et Flash CS3 sont les deux environnements auteurs permettant de produire du contenu Flash. Notons qu'il est aussi possible d'utiliser l'environnement Eclipse et les SDK de Flex et AIR.

- Les frameworks : Flex est un framework conçu pour faciliter le développement d'applications riches. Grâce à AIR, ces applications peuvent sortir du navigateur et être déployées sur le bureau.

Chaque outil de développement ou framework répond à une attente spécifique et correspond à un profil type.

Un graphiste-développeur sera intéressé par le développement d'interfaces animées et de sites Internet au sein de Flash CS3. La connaissance d'un environnement comme Flash CS3 lui permettra aussi de développer des composants réutilisables au sein de Flex et AIR.

Un développeur préférera peut être un environnement de développement comme celui offert par Flex Builder. Les applications produites grâce au framework Flex seront de nature différente et pourront facilement être déployées sur le bureau grâce à AIR.

---

Notons que les exemples présents dans cet ouvrage peuvent être utilisés au sein de Flash CS3, Flex et AIR.

---

Afin d'entamer cette aventure au cœur d'ActionScript 3, nous allons nous intéresser tout d'abord aux différentes nouveautés du langage.

En route pour le prochain chapitre intitulé *Langage et API* !

# 2

## Langage et API

<b>LE LANGAGE ACTIONSCRIPT 3.....</b>	<b>1</b>
MACHINES VIRTUELLES .....	4
TRADUCTION DYNAMIQUE .....	5
GESTION DES TYPES A L'EXECUTION .....	6
ERREURS A L'EXECUTION.....	11
NOUVEAUX TYPES PRIMITIFS .....	13
VALEURS PAR DEFAUT .....	18
NOUVEAUX TYPES COMPOSITES .....	20
NOUVEAUX MOTS-CLES .....	20
FONCTIONS .....	21
CONTEXTE D'EXECUTION.....	24
BOUCLES.....	25
ENRICHISSEMENT DE LA CLASSE ARRAY.....	26
RAMASSE-MIETTES .....	29
BONNES PRATIQUES .....	32
AUTRES SUBTILITES .....	33

### Le langage ActionScript 3

Le langage ActionScript 3 intègre de nombreuses nouveautés que nous allons traiter tout au long de cet ouvrage. Ce chapitre va nous permettre de découvrir les nouvelles fonctionnalités et comportements essentiels à tout développeur ActionScript 3.

Avant de détailler les nouveautés liées au langage ActionScript 3, il convient de définir tout d'abord ce que nous entendons par le terme ActionScript.

De manière générale, le terme ActionScript englobe deux composantes importantes :

- Le cœur du langage : il s'agit du langage ActionScript basé sur la spécification *ECMAScript* (ECMA-262) et intègre partiellement certaines fonctionnalités issues de la spécification *ECMAScript* 4.
- L'API du lecteur Flash : il s'agit des fonctionnalités du lecteur Flash. Toutes les classes nécessitant d'être importées font partie de l'API du lecteur et non du cœur du langage ActionScript.

Ainsi, l'interface de programmation du lecteur ou le langage peuvent être mise à jour indépendamment.

Le lecteur Flash 10 devrait normalement intégrer une gestion de la 3D native ainsi qu'une implémentation plus complète de la spécification *ECMAScript* 4. La gestion de la 3D concerne ici uniquement l'interface de programmation du lecteur Flash, à l'inverse les nouveaux objets définis par la spécification *ECMAScript* 4 sont directement liés au cœur du langage ActionScript.

D'un côté réside le langage ActionScript 3, de l'autre l'API du lecteur appelée généralement *interface de programmation*.

La figure 2-1 illustre les deux entités :



*Figure 2-1. Langage ActionScript 3.*

Contrairement aux précédentes versions d'ActionScript, nous remarquons qu'en ActionScript 3, les différentes fonctionnalités du lecteur Flash sont désormais stockées dans des paquetages spécifiques.

Afin d’afficher une vidéo nous utiliserons les objets issus du paquetage `flash.media`. A l’inverse, pour nous connecter à un serveur, nous utiliserons les objets issus du paquetage `flash.net`.

Flash CS3 est configuré afin d’importer automatiquement toutes les classes issues de l’API du lecteur Flash. Il n’est donc pas nécessaire d’importer manuellement les classes lorsque nous codons au sein de l’environnement auteur.

Un fichier *`implicitImports.xml`* situé au sein du répertoire d’installation de Flash CS3 (`C:\Program Files\Adobe\Adobe Flash CS3\fr\Configuration\ActionScript 3.0`) contient toutes les définitions de classe à importer :

```
<implicitImportsList>
  <implicitImport name = "flash.accessibility.*"/>
  <implicitImport name = "flash.display.*"/>
  <implicitImport name = "flash.errors.*"/>
  <implicitImport name = "flash.events.*"/>
  <implicitImport name = "flash.external.*"/>
  <implicitImport name = "flash.filters.*"/>
  <implicitImport name = "flash.geom.*"/>
  <implicitImport name = "flash.media.*"/>
  <implicitImport name = "flash.net.*"/>
  <implicitImport name = "flash.printing.*"/>
  <implicitImport name = "flash.system.*"/>
  <implicitImport name = "flash.text.*"/>
  <implicitImport name = "flash.ui.*"/>
  <implicitImport name = "flash.utils.*"/>
  <implicitImport name = "flash.xml.*"/>
</implicitImportsList>
```

Afin de créer un clip dynamiquement nous pouvons écrire directement sur une image du scénario :

```
| var monClip:MovieClip = new MovieClip();
```

Si nous plaçons notre code à l’extérieur de Flash au sein de classes, nous devons explicitement importer les classes nécessaires :

```
| import flash.display.MovieClip;
| var monClip:MovieClip = new MovieClip();
```

Dans cet ouvrage nous n’importerons pas les classes du lecteur lorsque nous programmerons dans l’environnement auteur de Flash. A l’inverse dès l’introduction des classes au sein du chapitre 8 intitulé *`Programmation orientée objet`*, nous importerons explicitement les classes utilisées.

---

## A retenir

---



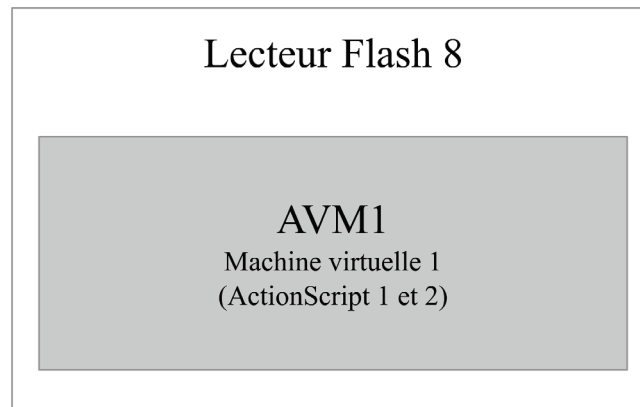
- Le langage `ActionScript 3` englobe deux composantes : le cœur du langage `ActionScript` et l'interface de programmation du lecteur `Flash`.
- Le cœur du langage est défini par la spécification `ECMAScript`.

## Machines virtuelles

Le code `ActionScript` est interprété par une partie du lecteur `Flash` appelée *machine virtuelle*. C'est cette dernière qui se charge de retranscrire en langage machine le code binaire (*`ActionScript byte code`*) généré par le compilateur.

Les précédentes versions du lecteur `Flash` intégraient une seule machine virtuelle appelée `AVM1` afin d'interpréter le code `ActionScript 1` et `2`. En réalité, le code binaire généré par le compilateur en `ActionScript 1` et `2` était le même, c'est la raison pour laquelle nous pouvions faire cohabiter au sein d'un même projet ces deux versions du langage `ActionScript`.

La figure 2-2 illustre la machine virtuelle 1 (`AVM1`) présente dans le lecteur `Flash 8` :

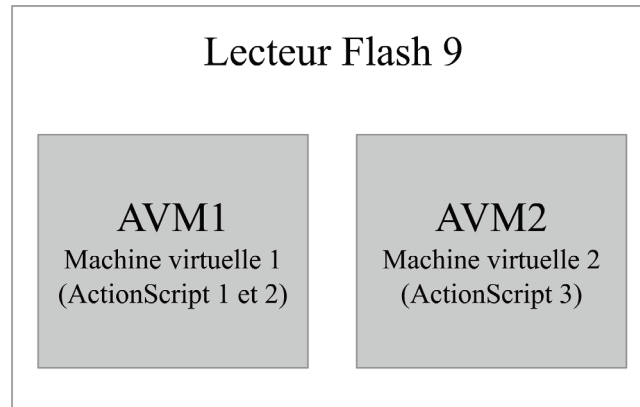


*Figure 2-2. AVM1 au sein du lecteur Flash 8.*

Le langage `ActionScript 3` n'est pas compatible avec cette première machine virtuelle, pour des raisons évidentes de rétrocompatibilité, le lecteur `Flash 9` embarque donc deux machines virtuelles.

Lors de la lecture d'un `SWF`, le lecteur sélectionne automatiquement la machine virtuelle appropriée afin d'interpréter le code `ActionScript` présent au sein du `SWF`. Ainsi, une application `ActionScript 1` et `2` sera interprétée au sein du lecteur `Flash 9` par la machine virtuelle 1 (`AVM1`) et ne bénéficiera d'aucune optimisation des performances.

La figure 2-3 présente les deux machines virtuelles au sein du lecteur Flash 9 :



*Figure 2-3. Le lecteur Flash 9 et les deux machines virtuelles AVM1 et AVM2.*

Seules les animations compilées en ActionScript 3 pourront bénéficier des optimisations réalisées par la nouvelle machine virtuelle (AVM2).

## A retenir

- Les précédentes versions du lecteur Flash intégraient une seule machine virtuelle afin d'interpréter le code ActionScript 1 et 2.
- La machine virtuelle 1 (AVM1) interprète le code ActionScript 1 et 2.
- La machine virtuelle 2 (AVM2) interprète seulement le code ActionScript 3.
- Le lecteur Flash intègre les deux machines virtuelles (AVM1 et AVM2).
- Le langage ActionScript 3 ne peut pas cohabiter avec les précédentes versions d'ActionScript.

## Traduction dynamique

Afin d'optimiser les performances, la machine virtuelle 2 (AVM2) du lecteur Flash 9 intègre un mécanisme innovant de compilation du code à la volée. Bien que le terme puisse paraître étonnant, ce principe appelé généralement *traduction dynamique* permet d'obtenir de meilleures performances d'exécution du code en compilant ce dernier à l'exécution.

Dans les précédentes versions du lecteur Flash, le code présent au sein du SWF était directement retranscrit par la machine virtuelle en

langage machine sans aucune optimisation liée à la plateforme en cours.

En ActionScript 3 la machine virtuelle retranscrit le code binaire (*ActionScript byte code*) en langage machine à l'aide d'un compilateur à la volée appelé couramment *compilateur à la volée*. (Just-in-time compiler). Ce dernier permet de compiler uniquement le code utilisé et de manière optimisée selon la plateforme en cours.

La machine virtuelle peut donc optimiser les instructions pour un processeur spécifique tout en prenant en considération les différentes contraintes de la plateforme.

Pour plus d'informations liées à la compilation à l'exécution, rendez vous à l'adresse suivante :

[http://en.wikipedia.org/wiki/Just-in-time\\_compilation](http://en.wikipedia.org/wiki/Just-in-time_compilation)

[http://fr.wikipedia.org/wiki/Compilation\\_%C3%A0\\_la\\_vol%C3%A9e](http://fr.wikipedia.org/wiki/Compilation_%C3%A0_la_vol%C3%A9e)

## Gestion des types à l'exécution

ActionScript 2 introduit au sein de Flash MX 2004 la notion de typage fort. Cela consistait à associer un type de données à une variable à l'aide de la syntaxe suivante :

```
| variable:Type
```

Dans le code suivant, nous tentions d'affecter une chaîne à une variable de type `Number` :

```
| var distance:Number = "150";
```

L'erreur suivante était générée à la compilation :

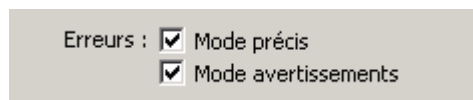
```
| Incompatibilité de types dans l'instruction d'affectation : String détecté au lieu de Number.
```

En ActionScript 3, nous bénéficions du même mécanisme de vérification des types à la compilation. En compilant le même code en ActionScript 3, l'erreur suivante est générée :

```
| 1067: Contrainte implicite d'une valeur du type String vers un type sans rapport Number.
```

Ce comportement est appelé *Mode précis* dans Flash CS3 et peut être désactivé par l'intermédiaire du panneau *Paramètres d'ActionScript 3.0*. A travers le panneau *Paramètres de publication*, puis de l'onglet *Flash*, nous cliquons sur le bouton *Paramètres*.

Nous obtenons un panneau *Paramètres d'ActionScript 3* contenant deux options liées aux erreurs comme l'illustre la figure 2-4 :



*Figure 2-4. Options du compilateur ActionScript 3.*

Nous remarquons que par défaut, le *Mode précis* est activé, nous reviendrons très vite sur le *Mode avertissements*.

En décochant la case *Mode précis*, nous désactivons la vérification des types à la compilation afin de découvrir un comportement extrêmement important apporté par ActionScript 3.

En testant le code suivant en mode non précis, nous remarquons qu'aucune erreur de compilation n'est générée :

```
| var distance:Number = "150";
```

A l'exécution, la machine virtuelle 2 (AVM2) convertit automatiquement la chaîne de caractères 150 en un nombre entier de type *int*.

Afin de vérifier cette conversion automatique, nous pouvons utiliser la fonction *describeType* du paquetage *flash.utils* :

```
| var distance:Number = "150";  
/* affiche :  
<type name="int" base="Object" isDynamic="false" isFinal="true"  
isStatic="false">  
  <extendsClass type="Object"/>  
  <constructor>  
    <parameter index="1" type="*" optional="true"/>  
  </constructor>  
</type>  
*/  
| trace( describeType ( distance ) );
```

La fonction *describeType* renvoie un objet XML décrivant le type de la variable. Nous pouvons remarquer que l'attribut *name* du nœud *type* renvoie *int*.

En modifiant la chaîne de caractères nous obtenons une conversion automatique vers le type *Number* :

```
| var distance:Number = "150.5";  
/* affiche :  
<type name="Number" base="Object" isDynamic="false" isFinal="true"  
isStatic="false">  
  <extendsClass type="Object"/>  
  <constructor>  
    <parameter index="1" type="*" optional="true"/>  
  </constructor>  
</type>  
*/
```

```
| trace( describeType ( distance ) );
```

Si nous tentons d'affecter un autre type de données à celle-ci, la machine virtuelle 2 (AVM2) conserve le type `Number` et convertit implicitement les données à l'exécution.

---

Contrairement au mode précis, ce comportement de vérification des types à l'exécution ne peut pas être désactivé.

---

Nous pourrions ainsi en conclure de toujours conserver le mode précis afin de ne pas être surpris par ce comportement, mais certaines erreurs de types ne peuvent être détectées par le compilateur car celles-ci n'interviennent qu'à l'exécution.

Dans le code suivant, le compilateur ne détecte aucune erreur :

```
| var tableauDonnees:Array = [ "150", "250" ];  
// l'entrée du tableau est automatiquement convertie en int  
var distance:Number = tableauDonnees[0];
```

A l'exécution, la chaîne de caractères présente à l'index 0 est automatiquement convertie en `int`. Cette conversion reste silencieuse tant que celle-ci réussit, le cas échéant une erreur à l'exécution est levée.

Dans le code suivant, nous tentons de stocker une chaîne de caractères au sein d'une variable de type `MovieClip` :

```
| var tableauDonnees:Array = [ "clip", "250" ];  
// lève une erreur à l'exécution  
var clip:MovieClip = tableauDonnees[0];
```

A l'exécution, la machine virtuelle 2 (AVM2) tente de convertir la chaîne en `MovieClip` et échoue, l'erreur à l'exécution suivante est levée :

```
| TypeError: Error #1034: Echec de la contrainte de type : conversion de "clip" en  
flash.display.MovieClip impossible.
```

Nous pouvons alors nous interroger sur l'intérêt d'un tel comportement, pourquoi la machine virtuelle s'évertue-t-elle à conserver les types à l'exécution et convertit automatiquement les données ?

Afin de garantir des performances optimales, la machine virtuelle 2 (AVM2) s'appuie sur les types définis par le développeur. Ainsi, lorsque nous typons une variable, l'occupation mémoire est optimisée spécifiquement pour ce type, ainsi que les instructions processeur.

---

Il ne faut donc pas considérer ce comportement comme un désagrément mais comme un avantage contribuant à de meilleures performances.

---

Ce comportement diffère d'ActionScript 2, où la machine virtuelle 1 (AVM1) évaluait dynamiquement tous les types à l'exécution, aucune optimisation n'était réalisée. Le typage des variables n'était qu'une aide à la compilation.

La grande nouveauté liée à ActionScript 3 réside donc dans l'intérêt du typage à la compilation comme à l'exécution. En associant un type à une variable en ActionScript 3 nous bénéficions d'une vérification des types à la compilation et d'une optimisation des calculs réalisés par le processeur et d'une meilleure optimisation mémoire.

---

Il est donc primordial de *toujours* typer nos variables en ActionScript 3, les performances en dépendent très nettement. Nous typerons systématiquement nos variables durant l'ensemble de l'ouvrage.

---

Voici un exemple permettant de justifier cette décision :

Une simple boucle utilise une variable d'incrément *i* de type *int* :

```
var debut:Number = getTimer();
for ( var i:int = 0; i< 500000; i++ )
{
}
// affiche : 5
trace( getTimer() - debut );
```

La boucle nécessite 5 millisecondes pour effectuer 500 000 itérations.

Sans typage de la variable *i*, la boucle suivante nécessite 14 fois plus de temps à s'exécuter :

```
var debut:Number = getTimer();
for ( var i = 0; i< 500000; i++ )
{
}
// affiche : 72
trace( getTimer() - debut );
```

Dans le code suivant, la variable `prenom` ne possède pas de type spécifique, la machine virtuelle doit évaluer elle-même le type ce qui ralentit le temps d'exécution :

```
var debut:Number = getTimer();

var prenom = "Bobby";
var prenomRaccourci:String;

for ( var i:int = 0; i< 500000; i++ )
{
    prenomRaccourci = prenom.substr ( 0, 3 );
}

// affiche : 430
trace( getTimer() - debut );

// affiche : Bob
trace ( prenomRaccourci );
```

En typant simplement la variable `prenom` nous divisons le temps d'exécution de presque deux fois :

```
var debut:Number = getTimer();

var prenom:String = "Bobby";
var prenomRaccourci:String;

for ( var i:int = 0; i< 500000; i++ )
{
    prenomRaccourci = prenom.substr ( 0, 3 );
}

// affiche : 232
trace( getTimer() - debut );

// affiche : Bob
trace ( prenomRaccourci );
```

Au sein du panneau *Paramètres ActionScript 3*, nous pouvons apercevoir un deuxième mode de compilation appelé *Mode avertissements*. Ce dernier permet d'indiquer plusieurs types d'erreurs comme par exemple les erreurs liées à la migration de code.

Supposons que nous tentions d'utiliser la méthode `attachMovie` dans un projet ActionScript 3 :

```
| var ref:MovieClip = this.attachMovie ("clip", "monClip", 0);
```

Au lieu d'indiquer un simple message d'erreur, le compilateur nous renseigne que notre code n'est pas compatible avec ActionScript 3 et nous propose son équivalent.

Le code précédent génère donc le message d'erreur suivant à la compilation :

```
Warning: 1060: Problème de migration : la méthode 'attachMovie' n'est plus prise en charge. Si le nom de la sous-classe de MovieClip est A, utilisez var mc= new A(); addChild(mc). Pour plus d'informations, consultez la classe DisplayObjectContainer.
```

Le *Mode avertissements* permet d'avertir le développeur de certains comportements à l'exécution risquant de le prendre au dépourvu.

---

Attention, les avertissements n'empêchent ni la compilation du code ni son exécution, mais avertissent simplement que le résultat de l'exécution risque de ne pas être celui attendu.

---

Dans le code suivant, un développeur tente de stocker une chaîne de caractère au sein d'une variable de type `Boolean` :

```
var prenom:Boolean = "Bobby";
```

A la compilation, l'avertissement suivant est affiché :

```
Warning: 3590: String utilisée alors qu'une valeur booléenne est attendue. L'expression va être transtypée comme booléenne.
```

Il est donc fortement conseillé de conserver le *Mode précis* ainsi que le mode avertissements afin d'intercepter un maximum d'erreurs à la compilation.

Comme nous l'avons vu précédemment, le lecteur Flash 9 n'échoue plus en silence et lève des erreurs à l'exécution. Nous allons nous intéresser à ce nouveau comportement dans la partie suivante.

## A retenir

- Il est possible de désactiver la vérification de type à la compilation.
- Il n'est pas possible de désactiver la vérification de type à l'exécution.
- Le typage en ActionScript 2 se limitait à une aide à la compilation.
- Le typage en ActionScript 3 aide à la compilation et à l'exécution.
- Dans un souci d'optimisation des performances, il est fortement recommandé de typer les variables en ActionScript 3.
- Il est fortement conseillé de conserver le mode précis ainsi que le mode avertissements afin d'intercepter un maximum d'erreurs à la compilation.

## Erreurs à l'exécution

Nous avons traité précédemment des erreurs de compilation à l'aide du mode précis et abordé la notion d'erreurs à l'exécution.



Une des grandes nouveautés du lecteur Flash 9 réside dans la gestion des erreurs. Souvenez-vous, les précédentes versions du lecteur Flash ne levaient aucune erreur à l'exécution et échouaient en silence.

En ActionScript 3 lorsque l'exécution du programme est interrompue de manière anormale, on dit qu'une erreur d'exécution est levée.

Afin de générer une erreur à l'exécution nous pouvons tester le code suivant :

```
// définition d'une variable de type MovieClip
// sa valeur par défaut est null
var monClip:MovieClip;

// nous tentons de récupérer le nombre d'images du scénario du clip
// il vaut null, une erreur d' exécution est levée
var nbImages:int = monClip.totalFrames;
```

La fenêtre de sortie affiche l'erreur suivante :

```
TypeError: Error #1009: Il est impossible d'accéder à la propriété ou à la
méthode d'une référence d'objet nul.
```

Afin de gérer cette erreur, nous pouvons utiliser un bloc `try catch` :

```
// définition d'une variable de type MovieClip
// sa valeur par défaut est null
var monClip:MovieClip;

var nbImages:int;

// grâce au bloc try catch nous pouvons gérer l'erreur
try
{
    nbImages = monClip.totalFrames;
} catch ( pErreur:Error )
{
    trace ( "une erreur d'exécution a été levée !");
}
```

Bien entendu, nous n'utiliserons pas systématiquement les blocs `try catch` afin d'éviter d'afficher les erreurs à l'exécution. Certains tests simples, que nous découvrirons au cours de l'ouvrage, nous permettront quelque fois d'éviter d'avoir recours à ces blocs.

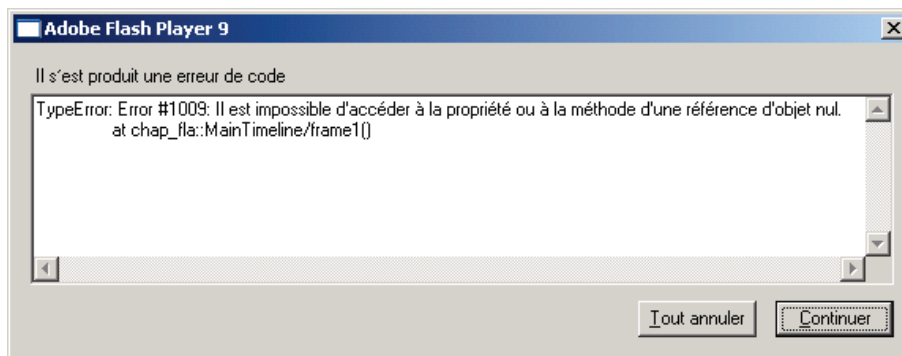
Dans un contexte d'erreurs à l'exécution, il convient de définir les deux déclinaisons du lecteur Flash existantes :

- Version de débogage (*Player Debug*) : cette version du lecteur est destinée au développement et affiche les erreurs à l'exécution en ouvrant une fenêtre spécifique indiquant l'erreur en cours. Ce lecteur est installé

automatiquement lors de l'installation de l'environnement de développement Flash CS3 ou Flex Builder 2 et 3.

- Version production (*Player Release*) : cette version du lecteur est disponible depuis le site d'Adobe. Les personnes n'ayant pas d'environnement de développement installé utilisent cette version du lecteur. Ce lecteur n'affiche pas les erreurs à l'exécution afin de ne pas interrompre l'expérience de l'utilisateur.

Avec le lecteur de débogage, les erreurs non gérées par un bloc `try catch` ouvrent un panneau d'erreur au sein du navigateur comme l'illustre la figure 2-5 :



*Figure 2-5. Exemple d'erreur à l'exécution.*

Lorsqu'une erreur est levée, l'exécution du code est alors mise en pause. Nous pouvons alors décider de continuer l'exécution du code bien qu'une erreur vienne d'être levée ou bien de stopper totalement l'exécution de l'application.

## A retenir

- Le lecteur Flash 9 lève des erreurs à l'exécution.
- Ces erreurs ouvrent avec le lecteur de débogage une fenêtre indiquant l'erreur au sein du navigateur.
- Toutes les méthodes de l'API du lecteur en ActionScript 3 peuvent lever des erreurs à l'exécution.
- Le lecteur Flash 9 n'échoue plus en silence, le débogage est donc facilité.

## Nouveaux types primitifs

En ActionScript 2, seul le type `Number` existait afin de définir un nombre, ainsi pour stocker un nombre entier ou décimal le type `Number` était utilisé :

```
var age:Number = 20;
var vitesse:Number = 12.8;
```

Aucune distinction n'était faite entre les nombres entiers, décimaux et non négatifs.

ActionScript 3 intègre désormais trois types afin de représenter les nombres :

- `int` : représente un nombre entier 32 bit (32 bit signed integer)
- `uint` : représente un nombre entier non signé 32 bit. (32 bit unsigned integer)
- `Number` : représente un nombre décimal 64 bit (64-bit IEEE 754 double-precision floating-point number)

Notons que les deux nouveaux types `int` et `uint` ne prennent pas de majuscule, contrairement au type `Number` déjà présent au sein d'ActionScript 2.

Une variable de type `int` peut contenir un nombre oscillant entre -2147483648 et 2147483648 :

```
// affiche : -2147483648
trace( int.MIN_VALUE );

// affiche : 2147483648
trace( int.MAX_VALUE );
```

Une variable de type `uint` peut contenir un nombre entier oscillant entre 0 et 4294967295 :

```
// affiche : 0
trace( uint.MIN_VALUE );

// affiche : 4294967295
trace( uint.MAX_VALUE );
```

Attention, la machine virtuelle ActionScript 3 conserve les types à l'exécution, si nous tentons de stocker un nombre à virgule flottante au sein d'une variable de type `int` ou `uint`, le nombre est automatiquement converti en entier par la machine virtuelle :

```
var age:int = 22.2;

// affiche : 22
trace ( age );
```

Notons, que la machine virtuelle arrondi à l'entier inférieur :

```
var age:int = 22.8;

// affiche : 22
trace ( age );
```

Cette conversion automatique assurée par la machine virtuelle s'avère beaucoup plus rapide que la méthode `floor` de la classe `Math`.

Dans le code suivant, nous arrondissons l'entier au sein d'une boucle à l'aide de la méthode `floor`, la boucle nécessite 111 millisecondes :

```
var distance:Number = 45.2;
var arrondi:Number;

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )
{
    arrondi = Math.floor ( distance );
}

// affiche : 111
trace( getTimer() - debut );
```

À présent, nous laissons la machine virtuelle gérer pour nous l'arrondi du nombre :

```
var distance:Number = 45.2;
var arrondi:int;

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )
{
    arrondi = distance;
}

// affiche : 8
trace( getTimer() - debut );

// affiche : 45
trace( arrondi );
```

Nous obtenons le même résultat en 8 millisecondes, soit un temps d'exécution presque 14 fois plus rapide.

---

Attention, cette astuce n'est valable uniquement dans le cas de nombres positifs.

---

Dans le code suivant, nous remarquons que la méthode `floor` de la classe `Math` ne renvoie pas la même valeur que la conversion en `int` par la machine virtuelle :

```
var distance:int = -3.2;

// affiche : -3
trace(distance);

var profondeur:Number = Math.floor (-3.2);

// affiche : -4
```

```
| trace( profondeur );
```

Partant du principe qu'une distance est toujours positive, nous pouvons utiliser le type `uint` qui offre dans ce cas précis des performances similaires au type `int` :

```
| var arrondi:uint;
```

Malheureusement, le type `uint` s'avère généralement beaucoup plus lent, dès lors qu'une opération mathématique est effectuée. En revanche, le type `Number` s'avère plus rapide que le type `int` lors de division.

Lors de la définition d'une boucle, il convient de toujours préférer l'utilisation d'une variable d'incrément de type `int` :

```
var debut:Number = getTimer();  
for ( var i:int = 0; i< 5000000; i++ )  
{  
}  
  
// affiche : 61  
trace( getTimer() - debut );
```

A l'inverse, si nous utilisons un type `uint`, les performances chutent de presque 400% :

```
var debut:Number = getTimer();  
for ( var i:uint = 0; i< 5000000; i++ )  
{  
}  
  
// affiche : 238  
trace( getTimer() - debut );
```

Gardez à l'esprit, qu'en cas d'hésitation, il est préférable d'utiliser le type `Number` :

```
var debut:Number = getTimer();  
for ( var i:Number = 0; i< 5000000; i++ )  
{  
}  
  
// affiche : 102  
trace( getTimer() - debut );
```

Nous obtenons ainsi un compromis en termes de performances entre le type `int` et `uint`.

---

De manière générale, il est préférable d'éviter le type `uint`.

---

La même optimisation peut être obtenue pour calculer l'arrondi supérieur. Nous préférons laisser la machine virtuelle convertir à l'entier inférieur puis nous ajoutons 1 :

```
var distance:Number = 45.2;
var arrondi:int;

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )
{
    arrondi = distance + 1;
}

// affiche : 12
trace( getTimer() - debut );

// affiche : 46
trace( arrondi );
```

En utilisant la méthode `ceil` de la classe `Math`, nous ralentissons les performances d'environ 300% :

```
var distance:Number = 45.2;
var arrondi:Number;

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )
{
    arrondi = Math.ceil ( distance );
}

// affiche : 264
trace( getTimer() - debut );

// affiche : 46
trace( arrondi );
```

Pour plus d'astuces liées à l'optimisation, rendez-vous à l'adresse suivante :

<http://lab.polygonal.de/2007/05/10/bitwise-gems-fast-integer-math/>

---

A retenir

---

- Le type `int` permet de représenter un nombre entier 32 bit.
- Le type `uint` permet de représenter un nombre entier 32 bit non négatif.
- Le type `Number` permet de représenter un nombre décimal 64 bit.
- Il est conseillé d'utiliser le type `int` pour les nombres entiers, son utilisation permet d'optimiser les performances.
- Il est déconseillé d'utiliser le type `uint`.
- En cas d'hésitation, il convient d'utiliser le type `Number`.

## Valeurs par défaut

Il est important de noter que les valeurs `undefined` et `null` ont un comportement différent en ActionScript 3. Désormais, une variable renvoie `undefined` uniquement lorsque celle-ci n'existe pas où lorsque nous ne la typons pas :

```
var prenom;  
  
// affiche : undefined  
trace( prenom );
```

Lorsqu'une variable est typée mais ne possède aucune valeur, une valeur par défaut lui est attribuée :

```
var condition:Boolean;  
var total:int;  
var couleur:uint;  
var resultat:Number;  
var personnage:Object;  
var prenom:String;  
var donnees:*;  
  
// affiche : false  
trace( condition );  
  
// affiche : 0  
trace( total );  
  
// affiche : 0  
trace( couleur );  
  
// affiche : NaN  
trace( resultat );  
  
// affiche : null  
trace( personnage );  
  
// affiche : null  
trace( prenom );  
  
// affiche : undefined  
trace( donnees );
```

Le tableau suivant illustre les différentes valeurs attribuées par défaut aux types de données :

Type de données	Valeur par défaut
Boolean	false
int	0
Number	NaN
Object	null
String	null
uint	0
Non typée (équivalent au type *)	undefined
Autres types	null

*Tableau 1. Valeurs par défaut associées aux types de données.*

De la même manière, si nous tentons d'accéder à une propriété inexistante au sein d'une instance de classe non dynamique telle `String`, nous obtenons une erreur à la compilation :

```
var prenom:String = "Bob";
// génère une erreur à la compilation
trace( prenom.proprieteInexistante );
```

Si nous tentons d'accéder à une propriété inexistante, au sein d'une instance de classe dynamique, le compilateur ne procède à aucune vérification et nous obtenons la valeur `undefined` pour la propriété ciblée :

```
var objet:Object = new Object();
// affiche : undefined
trace( objet.proprieteInexistante );
```

Attention, une exception demeure pour les nombres, qui ne peuvent être `null` ou `undefined`. Si nous typons une variable avec le type `Number`, la valeur par défaut est `NaN` :

```
var distance:Number;
// affiche : NaN
trace( distance );
```



En utilisant le type `int` ou `uint`, les variables sont automatiquement initialisées à 0 :

```
var distance:int;

var autreDistance:uint;

// affiche : 0
trace( distance );

// affiche : 0
trace( autreDistance );
```

## A retenir

- Une variable renvoie `undefined` uniquement lorsque celle-ci n'existe pas ou n'est pas typée.
- Lorsqu'une variable est typée mais ne possède aucune valeur, la machine virtuelle attribue automatiquement une valeur par défaut.

## Nouveaux types composites

Deux nouveaux types composites sont intégrés en ActionScript 3 :

- Les expressions régulières (`RegExp`) : elles permettent d'effectuer des recherches complexes sur des chaînes de caractères. Nous reviendrons sur les expressions régulières au cours de certains exercices.
- E4X (ECMAScript 4 XML) : la spécification ECMAScript 4 intègre un objet XML en tant qu'objet natif. Nous reviendrons sur le format XML et E4X au cours de certains exercices.

## Nouveaux mots-clés

Le mot clé `is` introduit par ActionScript 3 remplace l'ancien mot-clé `instanceof` des précédentes versions d'ActionScript.

Ainsi pour tester si une variable est d'un type spécifique nous utilisons le mot-clé `is` :

```
var tableauDonnees:Array = [5654, 95, 54, 687968, 97851];

// affiche : true
trace( tableauDonnees is Array );

// affiche : true
trace( tableauDonnees is Object );

// affiche : false
trace( tableauDonnees is MovieClip );
```

Un autre mot-clé nommé `as` fait aussi son apparition. Ce dernier permet de transtyper un objet vers un type spécifique.

Dans le code suivant, nous définissons une variable de type `DisplayObject`, mais celle-ci contient en réalité une instance de `MovieClip` :

```
| var monClip:DisplayObject = new MovieClip();
```

Si nous tentons d'appeler une méthode de la classe `MovieClip` sur la variable `monClip`, une erreur à la compilation est générée.

Afin de pouvoir appeler la méthode sans que le compilateur ne nous bloque, nous pouvons transtyper vers le type `MovieClip` :

```
| // transtypage en MovieClip  
| (monClip as MovieClip).gotoAndStop(2);
```

En cas d'échec du transtypage, le résultat du transtypage renvoie `null`, nous pouvons donc tester si le transtypage réussit de la manière suivante :

```
| var monClip:DisplayObject = new MovieClip();  
|  
| // affiche : true  
| trace( MovieClip(monClip) != null );
```

Nous aurions pu transtyper avec l'écriture traditionnelle suivante :

```
| var monClip:DisplayObject = new MovieClip();  
|  
| // transtypage en MovieClip  
| MovieClip(monClip).gotoAndStop(2);
```

En termes de performances, le mot clé `as` s'avère presque deux fois plus rapide. En cas d'échec lors du transtypage l'écriture précédente ne renvoie pas `null` mais lève une erreur à l'exécution.

## Fonctions

ActionScript 3 intègre de nouvelles fonctionnalités liées à la définition de fonctions. Nous pouvons désormais définir des paramètres par défaut pour les fonctions.

Prenons le cas d'une fonction affichant un message personnalisé :

```
| function alerte ( pMessage:String ):void  
| {  
|  
|     trace( pMessage );  
| }  
|
```

Cette fonction `alerte` accepte un paramètre accueillant le message à afficher. Si nous souhaitons l'exécuter nous devons obligatoirement passer un message :

```
| alerte ("voici un message d'alerte !");
```

Si nous omettons le paramètre :

```
// génère une erreur à la compilation  
alerte ();
```

L'erreur à la compilation suivante est générée :

```
1136: Nombre d'arguments incorrect. 1 attendus.
```

ActionScript 3 permet de définir une valeur par défaut pour le paramètre :

```
function alerte ( pMessage:String="message par défaut" ):void  
{  
    trace( pMessage );  
}
```

Une fois définie, nous pouvons appeler la fonction `alerte` sans passer de paramètres :

```
function alerte ( pMessage:String="message par défaut" ):void  
{  
    trace( pMessage );  
}  
  
// affiche : message par défaut  
alerte ();
```

Lorsque nous passons un paramètre spécifique, celui-ci écrase la valeur par défaut :

```
function alerte ( pMessage:String="message par défaut" ):void  
{  
    trace( pMessage );  
}  
  
// affiche : un message personnalisé !  
alerte ( "un message personnalisé !" );
```

En plus de cela, ActionScript 3 intègre un nouveau mécanisme lié aux paramètres aléatoires.

Imaginons que nous devons créer une fonction pouvant accueillir un nombre aléatoire de paramètres. En ActionScript 1 et 2, nous ne pouvions l'indiquer au sein de la signature de la fonction.

Nous définissons donc une fonction sans paramètre, puis nous utilisons le tableau `arguments` regroupant l'ensemble des paramètres :

```
function calculMoyenne ():Number
{
    var lng:Number = arguments.length;
    var total:Number = 0;

    for (var i:Number = 0; i< lng; i++)
    {
        total += arguments[i];
    }

    return total / lng;
}

var moyenne:Number = calculMoyenne ( 50, 48, 78, 20, 90 );

// affiche : 57.2
trace( moyenne );
```

Bien que cette écriture puisse paraître très souple, elle posait néanmoins un problème de relecture du code. En relisant la signature de la fonction, un développeur pouvait penser que la fonction `calculMoyenne` n'acceptait aucun paramètre, alors que ce n'était pas le cas.

Afin de résoudre cette ambiguïté, ActionScript 3 introduit un mot-clé permettant de spécifier dans les paramètres que la fonction en cours reçoit un nombre variable de paramètres.

Pour cela nous ajoutons trois points de suspensions en tant que paramètre de la fonction, suivi d'un nom de variable de notre choix.

Le même code s'écrit donc de la manière suivante en ActionScript 3 :

```
function calculMoyenne ( ...parametres ):Number
{
    var lng:int = parametres.length;
    var total:Number = 0;

    for (var i:Number = 0; i< lng; i++)
    {
        total += parametres[i];
    }

    return total / lng;
}

var moyenne:Number = calculMoyenne ( 50, 48, 78, 20, 90 );
```

```
// affiche : 57.2  
trace( moyenne );
```

En relisant le code précédent, le développeur ActionScript 3 peut facilement détecter les fonctions accueillant un nombre de paramètres aléatoires.

## Contexte d'exécution

Afin que vous ne soyez pas surpris, il convient de s'attarder quelques instants sur le nouveau comportement des fonctions passées en référence.

Souvenez-vous, en ActionScript 1 et 2, nous pouvions passer en référence une fonction, celle-ci perdait alors son contexte d'origine et épousait comme contexte le nouvel objet :

```
var personnage:Object = { age : 25, nom : "Bobby" };  
  
// la fonction parler est passée en référence  
personnage.parler = parler;  
  
function parler ( )  
{  
    trace("bonjour, je m'appelle " + this.nom + ", j'ai " + this.age + " ans");  
}  
  
// appel de la méthode  
// affiche : bonjour, je m'appelle Bobby, j'ai 25 ans  
personnage.parler();
```

En ActionScript 3, la fonction `parler` conserve son contexte d'origine et ne s'exécute donc plus dans le contexte de l'objet `personnage` :

```
var personnage:Object = { age : 25, nom : "Bobby" };  
  
// la fonction parler est passée en référence  
personnage.parler = parler;  
  
function parler ( )  
{  
    trace("bonjour, je m'appelle " + this.nom + ", j'ai " + this.age + " ans");  
}  
  
// appel de la méthode  
// affiche : bonjour, je m'appelle undefined, j'ai undefined ans  
personnage.parler();
```

De nombreux développeurs ActionScript se basaient sur ce changement de contexte afin de réutiliser des fonctions.

Nous devons donc garder à l'esprit ce nouveau comportement apporté par ActionScript 3 durant l'ensemble de l'ouvrage.

## Boucles

ActionScript 3 introduit une nouvelle boucle permettant d'itérer au sein des propriétés d'un objet et d'accéder directement au contenu de chacune d'entre elles.

Dans le code suivant, nous affichons les propriétés de l'objet `personnage` à l'aide d'une boucle `for in` :

```
var personnage:Object = { prenom : "Bobby", age : 50 };  
for (var p:String in personnage)  
{  
    /* affiche :  
    age  
    prenom  
    */  
    trace( p );  
}
```

Attention, l'ordre d'énumération des propriétés peut changer selon les machines. Il est donc essentiel de ne pas se baser sur l'ordre d'énumération des propriétés.

Notons que la boucle `for in` en ActionScript 3 ne boucle plus de la dernière entrée à la première comme c'était le cas en ActionScript 1 et 2, mais de la première à la dernière.

Nous pouvons donc désormais utiliser la boucle `for in` afin d'itérer au sein d'un tableau sans se soucier du fait que la boucle parte de la fin du tableau :

```
var tableauDonnees:Array = [ 5654, 95, 54, 687968, 97851];  
for ( var p:String in tableauDonnees )  
{  
    /* affiche :  
    5654  
    95  
    54  
    687968  
    97851  
    */  
    trace( tableauDonnees[p] );  
}
```

La boucle `for each` accède elle, directement au contenu de chaque propriété :

```
var personnage:Object = { prenom : "Bobby", age : 50 };  
for each ( var p:* in personnage )  
{  
    /* affiche :  
    50  
    Bobby  
    */  
    trace( p );  
}
```

Nous pouvons donc plus simplement itérer au sein du tableau à l'aide de la nouvelle boucle `for each` :

```
var tableauDonnees:Array = [ 5654, 95, 54, 687968, 97851 ];  
for each ( var p:* in tableauDonnees )  
{  
    /* affiche :  
    5654  
    95  
    54  
    687968  
    97851  
    */  
    trace( p );  
}
```

Nous allons nous intéresser dans la prochaine partie au concept de ramasse-miettes qui s'avère très important en ActionScript 3.

## Enrichissement de la classe Array

La classe `Array` bénéficie de nouvelles méthodes en ActionScript 3 facilitant la manipulation de données.

La méthode `forEach` permet d'itérer simplement au sein du tableau :

```
// Array.forEach procède à une navigation simple  
// sur chaque élément à l'aide d'une fonction spécifique  
var prenom:Array = [ "bobby", "willy", "ritchie" ];  
function navigue ( element:*, index:int, tableau:Array ):void  
{  
    trace ( element + " : " + index + " : " + tableau);  
}
```

```
/* affiche :  
bobby : 0 : bobby,willy,ritchie  
willy : 1 : bobby,willy,ritchie  
ritchie : 2 : bobby,willy,ritchie  
*/  
prenoms.forEach( navigue );
```

Toutes ces nouvelles méthodes fonctionnent sur le même principe. Une fonction de navigation est passée en paramètre afin d’itérer et de traiter les données au sein du tableau.

La méthode **every** exécute la fonction de navigation jusqu’à ce que celle ci ou l’élément parcouru renvoient **false**.

Il est donc très simple de déterminer si un tableau contient des valeurs attendues. Dans le code suivant, nous testons si le tableau **donnees** contient uniquement des nombres :

```
var donnees:Array = [ 12, "bobby", "willy", 58, "ritchie" ];  
function navigue ( element:*, index:int, tableau:Array ):Boolean  
{  
    return ( element is Number );  
}  
var tableauNombres:Boolean = donnees.every ( navigue );  
// affiche : false  
trace( tableauNombres );
```

La méthode **map** permet la création d’un tableau relatif au retour de la fonction de navigation. Dans le code suivant, nous appliquons un formatage aux données du tableau **prenoms**.

Un nouveau tableau de prénoms formatés est créé :

```
var prenoms:Array = ["bobby", "willy", "ritchie"];  
function navigue ( element:*, index:int, tableau:Array ):String  
{  
    return element.charAt(0).toUpperCase()+element.substr(1).toLowerCase();  
}  
// on crée un tableau à partir du retour de la fonction navigue  
var prenomsFormates:Array = prenoms.map ( navigue );  
// affiche : Bobby,Willy,Ritchie  
trace( prenomsFormates );
```

La méthode **map** ne permet pas de filtrer les données. Toutes les données du tableau source sont ainsi placées au sein du tableau généré.



Si nous souhaitons filtrer les données, nous pouvons appeler la méthode `filter`. Dans le code suivant nous filtrons les utilisateurs mineurs et obtenons un tableau d'utilisateurs majeurs :

```
var utilisateurs:Array = [ { prenom : "Bobby", age : 18 },
                           { prenom : "Willy", age : 20 },
                           { prenom : "Ritchie", age : 16 },
                           { prenom : "Stevie", age : 15 } ];

function navigue ( element:*, index:int, tableau:Array ):Boolean
{
    return ( element.age >= 18 );
}

var utilisateursMajeurs:Array = utilisateurs.filter ( navigue );

function parcourir ( element:*, index:int, tableau:Array ):void
{
    trace ( element.prenom, element.age );
}

/* affiche :
Bobby 18
Willy 20
*/
utilisateursMajeurs.forEach( parcourir );
```

La méthode `some` permet de savoir si un élément existe au moins une fois au sein du tableau. La fonction de navigation est exécutée jusqu'à ce que celle-ci ou un élément du tableau renvoient `true` :

```
var utilisateurs:Array = [ { prenom : "Bobby", age : 18, sexe : "H" },
                           { prenom : "Linda", age : 18, sexe : "F" },
                           { prenom : "Ritchie", age : 16, sexe : "H" },
                           { prenom : "Stevie", age : 15, sexe : "H" } ]

function navigue ( element:*, index:int, tableau:Array ):Boolean
{
    return ( element.sexe == "F" );
}

// y'a t'il une femme au sein du tableau d'utilisateurs ?
var resultat:Boolean = utilisateurs.some ( navigue );

// affiche : true
trace( resultat );
```

Les méthodes `indexOf` et `lastIndexOf` font elles aussi leur apparition au sein de la classe `Array`, celles-ci permettent de rechercher si un élément existe et d'obtenir sa position :

```
var utilisateurs:Array = [ "Bobby", "Linda", "Ritchie", "Stevie", "Linda" ];  
  
var position:int = utilisateurs.indexOf ("Linda");  
  
var positionFin:int = utilisateurs.lastIndexOf ("Linda");  
  
// affiche : 1  
trace( position );  
  
// affiche : 4  
trace( positionFin );
```

Les méthodes `indexOf` et `lastIndexOf` permettent de rechercher un élément de type primitif mais aussi de type composite.

Nous pouvons ainsi rechercher la présence de références au sein d'un tableau :

```
var monClip:DisplayObject = new MovieClip();  
  
var monAutreClip:DisplayObject = new MovieClip();  
  
// une référence au clip monClip est placée au sein du tableau  
var tableauReferences:Array = [ monClip ];  
  
var position:int = tableauReferences.indexOf (monClip);  
  
var autrePosition:int = tableauReferences.lastIndexOf (monAutreClip);  
  
// affiche : 0  
trace( position );  
  
// affiche : -1  
trace( autrePosition );
```

Nous reviendrons sur certaines de ces méthodes au sein de l'ouvrage.

## A retenir

- Pensez à utiliser les nouvelles méthodes de la classe `Array` afin de traiter plus facilement les données au sein d'un tableau.

## Ramasse-miettes

Tout au long de l'ouvrage nous reviendrons sur le concept de *ramasse-miettes*. Bien que le terme puisse paraître fantaisiste, ce mécanisme va s'avérer extrêmement important durant nos développements ActionScript 3.

Pendant la durée de vie d'un programme, certains objets peuvent devenir inaccessibles. Afin, de ne pas saturer la mémoire, un mécanisme de suppression des objets inutilisés est intégré au sein du

lecteur Flash. Ce mécanisme est appelé ramasse-miettes ou plus couramment *Garbage collector* en Anglais.

Afin de bien comprendre ce mécanisme, nous définissons un simple objet référencé par une variable `personnage` :

```
| var personnage:Object = { prenom : "Bobby", age : 50 };
```

Pour rendre cet objet inaccessible et donc éligible à la suppression par le ramasse-miettes, nous pourrions être tentés d'utiliser le mot clé `delete` :

```
| var personnage:Object = { prenom : "Bobby", age : 50 };  
| // génère une erreur à la compilation  
| delete personnage;
```

Le code précédent, génère l'erreur de compilation suivante :

```
| 1189: Tentative de suppression de la propriété fixe personnage. Seules les  
| propriétés définies dynamiquement peuvent être supprimées.
```

Cette erreur traduit une modification du comportement du mot clé `delete` en ActionScript 3, qui ne peut être utilisé que sur des propriétés dynamiques d'objets dynamiques.

Ainsi, le mot clé `delete` pourrait être utilisé pour supprimer la propriété `prenom` au sein de l'objet `personnage` :

```
| var personnage:Object = { prenom : "Bobby", age : 50 };  
| // affiche : Bobby  
| trace( personnage.prenom );  
|  
| // supprime la propriété prenom  
| delete ( personnage.prenom );  
|  
| // affiche : undefined  
| trace( personnage.prenom );
```

Afin de supprimer correctement une référence, nous devons affecter la valeur `null` à celle-ci :

```
| var personnage:Object = { prenom : "Bobby", age : 50 };  
|  
| // supprime la référence vers l'objet personnage  
| personnage = null;
```

Lorsque l'objet ne possède plus aucune référence, nous pouvons estimer que l'objet est *éligible à la suppression*.

Nous devons donc toujours veiller à ce qu'aucune référence ne subsiste vers notre objet, au risque de le voir conservé en mémoire.

---

Attention, l'affectation de la valeur `null` à une référence ne déclenche en aucun cas le passage du

---

ramasse-miettes. Nous rendons simplement l'objet éligible à la suppression.

Il est important de garder à l'esprit que le passage du ramasse-miettes reste *potentiel*. L'algorithme de nettoyage effectué par ce dernier étant relativement gourmand en termes de ressources, son déclenchement reste limité au cas où le lecteur utiliserait trop de mémoire.

---

Dans le code suivant, nous supprimons une des deux références seulement :

```
var personnage:Object = { prenom : "Bobby", age : 50 };  
  
// copie d'une référence au sein du tableau  
var tableauPersonnages:Array = [ personnage ];  
  
// suppression d'une seule référence  
personnage = null;
```

Une autre référence vers l'objet *personnage* subsiste au sein du tableau, l'objet ne sera donc jamais supprimé par le ramasse-miettes :

```
var personnage:Object = { prenom : "Bobby", age : 50 };  
  
// copie d'une référence au sein du tableau  
var tableauPersonnages:Array = [ personnage ];  
  
// supprime une des deux références seulement  
personnage = null;  
  
var personnageOrigine:Object = tableauPersonnages[0];  
  
// affiche : Bobby  
trace( personnageOrigine.prenom );  
  
// affiche : 50  
trace( personnageOrigine.age );
```

Nous devons donc aussi supprimer la référence présente au sein du tableau afin de rendre l'objet *personnage* éligible à la suppression :

```
var personnage:Object = { prenom : "Bobby", age : 50 };  
  
// copie d'une référence au sein du tableau  
var tableauPersonnages:Array = [ personnage ];  
  
// supprime la première référence  
personnage = null;  
  
// supprime la seconde référence  
tableauPersonnages[0] = null;
```

Si la situation nous le permet, un moyen plus radical consiste à écraser le tableau contenant la référence :

```
var personnage:Object = { prenom : "Bobby", age : 50 };
```

---

```
// copie d'une référence au sein du tableau
var tableauPersonnages:Array = [ personnage ];

// supprime la première référence
personnage = null;

// écrase le tableau stockant la référence
tableauPersonnages = new Array();
```

Depuis la version 9.0.115 du lecteur Flash 9, il est possible de déclencher le ramasse-miettes manuellement à l'aide de la méthode `gc` de la classe `System`. Notons que cette fonctionnalité n'est accessible qu'au sein du lecteur de débogage.

Nous reviendrons sur cette méthode au cours du prochain chapitre intitulé *Le modèle événementiel*.

## A retenir

- Il est possible de déclencher manuellement le ramasse-miettes au sein du lecteur de débogage 9.0.115.
- Afin qu'un objet soit éligible à la suppression par le ramasse-miettes nous devons passer toutes ses références à `null`.
- Le ramasse-miettes intervient lorsque la machine virtuelle juge cela nécessaire.

## Bonnes pratiques

Durant l'ensemble de l'ouvrage nous ferons usage de certaines bonnes pratiques, dont voici le détail :

Nous typerons les variables systématiquement afin d'optimiser les performances de nos applications et garantir une meilleure gestion des erreurs à la compilation.

Lors de la définition de boucles, nous utiliserons toujours une variable de référence afin d'éviter que la machine virtuelle ne réévalue la longueur du tableau à chaque itération.

Nous préférons donc le code suivant :

```
var tableauDonnees:Array = [ 5654, 95, 54, 687968, 97851];

// stockage de la longueur du tableau
var lng:int = tableauDonnees.length;

for ( var i:int = 0; i< lng; i++ )
{
}

}
```

A cette écriture non optimisée :

```
var tableauDonnees:Array = [ 5654, 95, 54, 687968, 97851];  
for ( var i:int = 0; i< tableauDonnees.length; i++ )  
{  
}
```

De la même manière, nous éviterons de redéfinir nos variables au sein des boucles.

Nous préférons l'écriture suivante :

```
var tableauDonnees:Array = [5654, 95, 54, 687968, 97851];  
var lng:int = tableauDonnees.length;  
// déclaration de la variable elementEnCours une seule et unique fois  
var elementEnCours:int;  
for ( var i:int = 0; i< lng; i++ )  
{  
    elementEnCours = tableauDonnees[i];  
}
```

A l'écriture suivante non optimisée :

```
var tableauDonnees:Array = [5654, 95, 54, 687968, 97851];  
for ( var i:int = 0; i< tableauDonnees.length; i++ )  
{  
    // déclaration de la variable elementEnCours à chaque itération  
    var elementEnCours:int = tableauDonnees[i];  
}
```

Découvrons à présent quelques dernières subtilités du langage ActionScript 3.

### Autres subtilités

Attention, le type `Void` existant en ActionScript 2, utilisé au sein des signatures de fonctions prend désormais un `v` minuscule en ActionScript 3.

Une fonction ActionScript 2 ne renvoyant aucune valeur s'écrivait de la manière suivante :

```
function initilisation ( ):Void  
{  
}
```

En ActionScript 3, le type `void` ne prend plus de majuscule :

```
function initialisation ( ):void
{
}

```

ActionScript 3 intègre un nouveau type de données permettant d'indiquer qu'une variable peut accueillir n'importe quel type de données.

En ActionScript 2 nous nous contentions de ne pas définir de type, en ActionScript 3 nous utilisons le type `*` :

```
var condition:* = true;
var total:* = 5;
var couleur:* = 0x990000;
var resultat:* = 45.4;
var personnage:* = new Object();
var prenom:* = "Bobby";

```

Nous utiliserons donc systématiquement le type `*` au sein d'une boucle `for each`, car la variable `p` doit pouvoir accueillir n'importe quel type de données :

```
var personnage:Object = { prenom : "Bobby", age : 50 };
for each ( var p:* in personnage )
{
    /* affiche :
    50
    Bobby
    */
    trace( p );
}

```

En utilisant un type spécifique pour la variable d'itération `p`, nous risquons de convertir implicitement les données itérées.

Dans le code suivant, nous utilisons le type `Boolean` pour la variable `p` :

```
var personnage:Object = { prenom : "Bobby", age : 50 };
for each ( var p:Boolean in personnage )
{
    /* affiche :
    true
    true
    */
    trace( p );
}

```

| }

Le contenu des propriétés est automatiquement converti en booléen. Au cas où une donnée ne pourrait être convertie en booléen, le code précédent pourrait lever une erreur à l'exécution.

Passons à présent aux nouveautés liées à l'interface de programmation du lecteur Flash.

Les deux grandes nouveautés du lecteur Flash 9 concernent la gestion de l'affichage ainsi que le modèle événementiel.

Au cours du prochain chapitre intitulé *Le modèle événementiel* nous allons nous intéresser à ce nouveau modèle à travers différents exercices d'applications. Nous apprendrons à maîtriser ce dernier et découvrirons comment en tirer profit tout au long de l'ouvrage.

Puis nous nous intéresserons au nouveau mécanisme de gestion de l'affichage appelée *Liste d'affichage* au cours du chapitre 4, nous verrons que ce dernier offre beaucoup plus de souplesse en termes de manipulation des objets graphiques et d'optimisation de l'affichage.

Vous êtes prêt pour la grande aventure ActionScript 3 ?



# 3

## Le modèle événementiel

<b>L'HISTOIRE .....</b>	<b>1</b>
<b>UN NOUVEAU MODELE EVENEMENTIEL.....</b>	<b>6</b>
TOUT EST ÉVÉNEMENTIEL .....	8
ECOUTER UN ÉVÉNEMENT .....	10
L'OBJET ÉVÉNEMENTIEL .....	13
LA CLASSE EVENT.....	16
<b>LES SOUS-CLASSES D'EVENT .....</b>	<b>17</b>
ARRÊTER L'ÉCOUTE D'UN ÉVÉNEMENT .....	19
MISE EN APPLICATION .....	20
<b>LA PUISSANCE DU COUPLAGE FAIBLE.....</b>	<b>25</b>
SOUPLESSE DE CODE .....	28
<b>ORDRE DE NOTIFICATION.....</b>	<b>30</b>
<b>REFERENCES FAIBLES.....</b>	<b>32</b>
<b>SUBTILITES.....</b>	<b>34</b>

### L'histoire

ActionScript 3 intègre un nouveau modèle événementiel que nous allons étudier ensemble tout au long de ce chapitre. Avant de l'aborder, rappelons-nous de l'ancien modèle apparu pour la première fois avec le lecteur Flash 6.

Jusqu'à maintenant nous définissions des fonctions que nous passions en référence à l'événement écouté. Prenons un exemple simple, pour écouter l'événement `onRelease` d'un bouton nous devons écrire le code suivant :

```
monBouton.onRelease = function ()
```

```
{  
  
    // affiche : _level0.monBouton  
    trace ( this );  
  
    this._alpha = 50;  
  
}
```

Une fonction anonyme était définie sur la propriété `onRelease` du bouton. Lorsqu'un clic souris intervenait sur le bouton, le lecteur Flash exécutait la fonction que nous avions définie et réduisait l'opacité du bouton de 50%. Cette écriture posait plusieurs problèmes.

Dans un premier temps, la fonction définie sur le bouton ne pouvait pas être réutilisée pour un autre événement, sur un objet différent. Pour pallier ce problème de réutilisation, certains développeurs faisaient appels à des références de fonctions afin de gérer l'événement :

```
function clicBouton ()  
{  
  
    // affiche : _level0.monBouton  
    trace (this);  
  
    this._alpha = 50;  
  
}  
  
monBouton.onRelease = clicBouton;
```

Cette écriture permettait de réutiliser la fonction `clicBouton` pour d'autres événements liés à différents objets, rendant le code plus aéré en particulier lors de l'écoute d'événements au sein de boucles.

---

Point important, le mot-clé `this` faisait ici référence au bouton et non au scénario sur lequel est définie la fonction `clicBouton`. Le contexte d'exécution d'origine de la fonction `clicBouton` était donc perdu lorsque celle-ci était passée en référence. La fonction épousait le contexte de l'objet auteur de l'événement.

---

Ensuite l'auto référence traduite par l'utilisation du mot-clé `this` était le seul moyen de faire référence à ce dernier. Ce comportement était quelque peu déroutant pour une personne découvrant `ActionScript`.

Voici un exemple mettant en évidence l'ambiguïté possible :

```
function clicBouton ()  
{  
  
    // affiche : _level0.monBouton
```

```
        trace ( this );  
  
        this._alpha = 50;  
  
    }  
  
    monBouton.onRelease = clicBouton;  
  
    clicBouton();
```

En exécutant la fonction `clicBouton` de manière traditionnelle le mot-clé `this` faisait alors référence au scénario sur lequel elle était définie, c'est à dire son contexte d'origine.

Le code suivant était peu digeste et rigide :

```
var ref:MovieClip;  
  
for ( var i:Number = 0; i< 50; i++ )  
{  
    ref = this.attachMovie ( "monClip", "monClip"+i, i );  
    ref.onRelease = function ()  
    {  
        trace("cliqué");  
    }  
}
```

Les développeurs préféraient donc l'utilisation d'une fonction séparée réutilisable :

```
var ref:MovieClip;  
  
for ( var i:Number = 0; i< 50; i++ )  
{  
    ref = this.attachMovie ( "monClip", "monClip"+i, i );  
    ref.onRelease = clicBouton;  
}  
  
function clicBouton ( )  
{  
    trace("cliqué");  
}
```

Le mécanisme était le même pour la plupart des événements. Macromedia, à l'époque de Flash MX (Flash 6) s'était rendu compte du manque de souplesse de ce système de fonctions définies sur les

événements, et intégra la notion d'écouteurs et de diffuseurs au sein du lecteur Flash 6. Les classes `Key`, `Selection`, `TextField` furent les premières à utiliser cette notion de diffuseurs écouteurs.

Pour écouter la saisie dans un champ texte, nous pouvions alors utiliser la syntaxe suivante :

```
var monEcouteur:Object = new Object();

monEcouteur.onChanged = function(pChamp:TextField)
{
    trace ( pChamp._name + " a diffusé l'événement onChanged" );
};

monChampTexte.addListener ( monEcouteur );
```

En appelant la méthode `addListener` sur notre champ texte nous souscrivions un objet appelé ici `monEcouteur` en tant qu'écouteur de l'événement `onChanged`. Une méthode du même nom devait être définie sur l'objet écouteur afin que celle-ci soit exécutée lorsque l'événement était diffusé.

Pour déterminer quelle touche du clavier était enfoncée, nous pouvions écrire le code suivant :

```
var monEcouteur:Object = new Object();

monEcouteur.onKeyDown = function()
{
    trace ( "La touche " + String.fromCharCode( Key.getCode() ) + " est enfoncée" );
};

Key.addListener ( monEcouteur );
```

Cette notion de méthode portant le nom de l'événement diffusé n'était pas évidente à comprendre et souffrait de la faiblesse suivante.

Aucun contrôle du nom de l'événement n'était effectué à la compilation ou à l'exécution. Le code suivant ne générait donc aucune erreur bien qu'une faute de frappe s'y soit glissée :

```
var monEcouteur:Object = new Object();

monEcouteur.onKeydown = function()
{
    trace ( "La touche " + String.fromCharCode( Key.getCode() ) + " est enfoncée" );
};
```

```
};  
  
Key.addListener ( monEcouteur );
```

Bien que critiquée, cette approche permettait néanmoins de souscrire un nombre illimité d'écouteurs auprès d'un événement, mais aussi de renvoyer des paramètres à la méthode écouteur.

---

Ces paramètres étaient généralement des informations liées à l'événement, par exemple l'auteur de l'événement diffusé.

---

En 2004, lors de la sortie du lecteur Flash 7 (Flash MX 2004), ActionScript 2 fit son apparition accompagné d'un lot de nouveaux composants appelés composants V2. Ces derniers étendaient le concept de diffuseurs écouteurs en intégrant une méthode `addEventListener` pour souscrire un écouteur auprès d'un événement. Contrairement à la méthode `addListener`, cette méthode stockait chaque écouteur dans des tableaux internes différents pour chaque événement.

En regardant l'évolution d'ActionScript au cours des dernières années, nous pouvons nous rendre compte du travail des ingénieurs visant à intégrer la notion d'écouteurs auprès de tous nouveaux objets créés. ActionScript 3 a été développé dans le but d'offrir un langage puissant et standard et prolonge ce concept en intégrant un nouveau modèle événementiel appelé « Document Object Model » (DOM3 Event Model) valable pour tous les objets liés à l'API du lecteur Flash 9.

Le W3C définit la spécification de ce modèle événementiel. Vous pouvez lire les dernières révisions à l'adresse suivante :

<http://www.w3.org/TR/DOM-Level-3-Events/>

Les développeurs ActionScript 2 ayant déjà utilisé la classe `EventDispatcher` seront ravis de savoir que toutes les classes de l'API du lecteur Flash héritent directement de celle-ci. Nous allons donc découvrir en profondeur le concept de diffuseurs écouteurs et voir comment cela améliore la clarté et la souplesse de notre code.

---

## A retenir

---

- L'ancien modèle événementiel a été entièrement revu en `ActionScript 3`.
- La grande puissance d'ActionScript 3 réside dans l'implémentation native d'`EventDispatcher`.

## Un nouveau modèle événementiel

En ActionScript 3, le modèle événementiel a clairement évolué. La manière dont les objets interagissent entre eux a été entièrement revue. Le principe même de ce modèle événementiel est tiré d'un modèle de conception ou « Design Pattern » appelé *Observateur* qui définit l'interaction entre plusieurs objets d'après un découpage bien spécifique.

Habituellement ce modèle de conception peut être défini de la manière suivante :

*« Le modèle de conception observateur définit une relation entre objets de type un à plusieurs, de façon que, lorsque un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement ».*

Le modèle de conception Observateur met en scène *trois* acteurs :

### Le sujet

Il est la source de l'événement, nous l'appellerons **sujet** car c'est lui qui diffuse les événements qui peuvent être écoutés ou non par les observateurs. Ce sujet peut être un bouton, ou un clip par exemple. Ne vous inquiétez pas, tous ces événements sont documentés, la documentation présente pour chaque objet les événements diffusés.

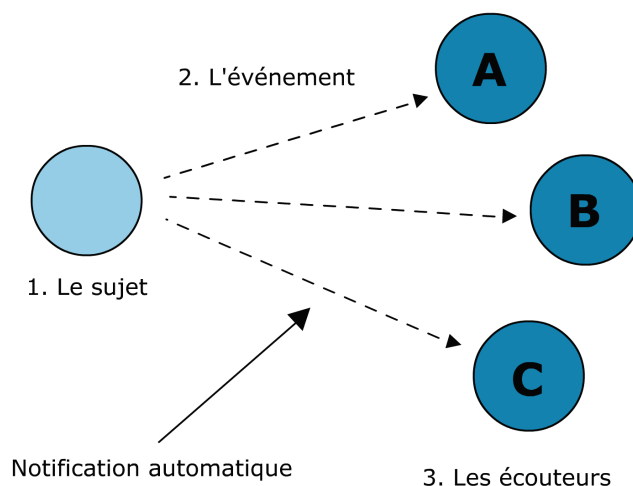
### L'événement

Nous pouvons conceptualiser la notion d'événement comme un changement d'état au sein du sujet. Un clic sur un bouton, la fin d'un chargement de données provoquera la diffusion d'un événement par l'objet sujet. Tous les événements existants en ActionScript 3 sont stockés dans des propriétés constantes de classes liées à l'événement.

### L'écouteur

L'écouteur a pour mission d'écouter un événement spécifique auprès d'un ou plusieurs sujets. Il peut être une fonction ou une méthode. Lorsque le sujet change d'état, ce dernier diffuse un événement approprié, si l'écouteur est souscrit auprès de l'événement diffusé, il en est notifié et exécute une action. Il est

important de souligner qu'il peut y avoir de multiples écouteurs pour un même événement. C'est justement l'un des points forts existant au sein de l'ancien modèle qui fut conservé dans ce nouveau représenté par la figure 3-1 :



*Figure 3-1. Schéma du modèle de conception  
Observateur*

Nos écouteurs sont dépendants du sujet, ils y ont *souscrit*. Le sujet ne connaît rien des écouteurs, il sait simplement s'il est observé ou non à travers sa liste interne d'écouteurs. Lorsque celui-ci change d'état un événement est *diffusé*, nos écouteurs en sont *notifiés* et peuvent alors réagir. Tout événement diffusé envoie des informations aux écouteurs leur permettant d'être tenus au courant des modifications et donc de rester à jour en permanence avec le sujet.

---

Nous découvrirons au fil de ces pages, la souplesse apportée par ce nouveau modèle événementiel, et vous apprécierez son fonctionnement très rapidement.

---

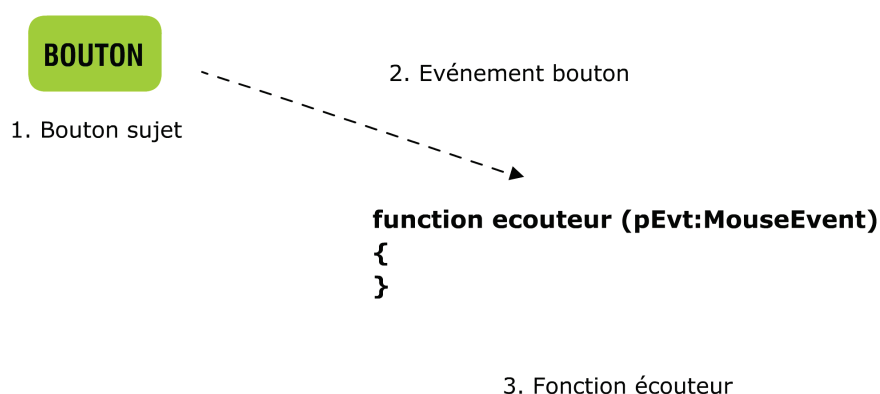
Cela reste encore relativement abstrait, pour y remédier voyons ensemble comment ce modèle événementiel s'établit au sein d'une application Flash traditionnelle.

Prenons un exemple simple constitué d'un bouton et d'une fonction écouteur réagissant lorsque l'utilisateur clique sur ce bouton. Nous avons dans ce cas nos trois acteurs mettant en scène le nouveau modèle événementiel ActionScript 3 :

- Le sujet : le bouton
- L'événement : le clic bouton

- L'écouteur : la fonction

Transposé dans une application ActionScript 3 traditionnelle, nous pouvons établir le schéma suivant :



*Figure 3-2. Modèle événementiel dans une application ActionScript 3*

Notre bouton est un sujet pouvant diffuser un événement. Celui-ci est écouté par une fonction écouteur.

## A retenir

- Il existe un seul et unique modèle événementiel en ActionScript 3.
- Ce modèle événementiel est basé sur le modèle de conception *Observateur*.
- Les trois acteurs de ce nouveau modèle événementiel sont : le sujet, l'événement, et l'écouteur.
- Nous pouvons avoir autant d'écouteurs que nous le souhaitons.
- Plusieurs écouteurs peuvent écouter le même événement.
- Un seul écouteur peut être souscrit à différents événements.

## Tout est événementiel

L'essentiel du modèle événementiel réside au sein d'une seule et unique classe appelée `EventDispatcher`. C'est elle qui offre la possibilité à un objet de diffuser des événements. Tous les objets issus de l'API du lecteur 9 et 10 héritent de cette classe, et possèdent de ce fait la capacité de diffuser des événements.



---

Souvenez-vous que toutes les classes issues du paquetage `flash` sont relatives à l'API du lecteur Flash.

---

Avant le lecteur 9, ces objets héritaient simplement de la classe `Object`, si nous souhaitions pouvoir diffuser des événements nous devons nous même implémenter la classe `EventDispatcher` au sein de ces objets.

Voici la chaîne d'héritage de la classe `MovieClip` avant le lecteur Flash 9 :

```
Object
|
+-MovieClip
```

Dans les précédentes versions du lecteur Flash, la classe `MovieClip` étendait directement la classe `Object`, si nous souhaitions pouvoir diffuser des événements à partir d'un `MovieClip` nous devons implémenter nous-mêmes un nouveau modèle événementiel. Voyons comme cela a évolué avec l'ActionScript 3 au sein du lecteur 9.

Depuis le lecteur 9 et ActionScript 3 :

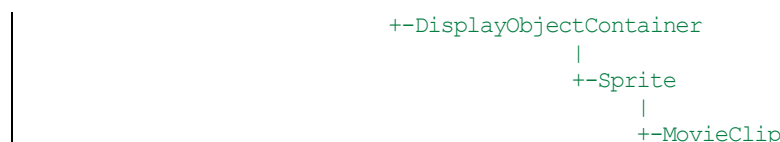
```
Object
|
+-EventDispatcher
```

La classe `EventDispatcher` hérite désormais de la classe `Object`, tous les autres objets héritent ensuite d'`EventDispatcher` et bénéficient donc de la diffusion d'événements en natif. C'est une grande évolution pour nous autres développeurs ActionScript, car la plupart des objets que nous manipulerons auront la possibilité de diffuser des événements par défaut.

Ainsi la classe `MovieClip` en ActionScript 3 hérite d'abord de la classe `Object` puis d'`EventDispatcher` pour ensuite hériter des différentes classes graphiques. Toutes les classes résidant dans le paquetage `flash` ne dérogent pas à la règle et suivent ce même principe.

Voici la chaîne d'héritage complète de la classe `MovieClip` en ActionScript 3 :

```
Object
|
+-EventDispatcher
  |
  +-DisplayObject
    |
    +-InteractiveObject
      |
      |
```



Voyons ensemble comment utiliser ce nouveau modèle événementiel à travers différents objets courants.

## Écouter un événement

Lorsque vous souhaitez être tenu au courant des dernières nouveautés de votre vidéoclub, vous avez plusieurs possibilités.

La première consiste à vous déplacer jusqu'au vidéoclub afin de découvrir les dernières sorties. Une approche classique qui ne s'avère pas vraiment optimisée. Nous pouvons déjà imaginer le nombre de visites que vous effectuerez dans le magasin pour vous rendre compte qu'aucun nouveau DVD ne vous intéresse.

Une deuxième approche consiste à laisser une liste de films que vous attendez au gérant du vidéoclub et attendre que celui-ci vous appelle lorsqu'un des films est arrivé. Une approche beaucoup plus efficace pour vous et pour le gérant qui en a assez de vous voir repartir déçu. Le gérant incarne alors le diffuseur, et vous-même devenez l'écouteur. Le gérant du vidéoclub ne sait rien sur vous si ce n'est votre nom et votre numéro de téléphone. Il est donc le sujet, vous êtes alors l'écouteur car vous êtes souscrit à un événement précis : « la disponibilité d'un film que vous recherchez ».

En ActionScript 3 les objets fonctionnent de la même manière. Pour obtenir une notification lorsque l'utilisateur clique sur un bouton, nous souscrivons un écouteur auprès d'un événement diffusé par le sujet, ici notre bouton.

Afin d'écouter un événement spécifique, nous utilisons la méthode `addEventListener` dont voici la signature :

```
addEventListener(type:String, listener:Function, useCapture:Boolean = false,
priority:int = 0, useWeakReference:Boolean = false):void
```

Le premier attend l'événement à écouter sous la forme d'une chaîne de caractères. Le deuxième paramètre prend en référence l'écouteur qui sera souscrit auprès de l'événement. Les trois derniers paramètres seront expliqués plus tard.

---

Souvenez-vous que seuls les objets résidant dans le paquetage `flash` utilisent ce modèle événementiel.

---

Le schéma à mémoriser est donc le suivant :

```
objet.addEventListener("evenement", ecouteur);
```

Voyons à l'aide d'un exemple simple, comment écouter un événement.

Dans un nouveau document Flash, créez un symbole bouton, et placez une occurrence de ce dernier sur la scène et nommez la `monBouton`.

Sur un calque AS tapez le code suivant :

```
monBouton.addEventListener ( "click", clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{
    // affiche : événement diffusé
    trace("événement diffusé");
}
```

Nous écoutons ici l'événement `click` sur le bouton `monBouton` et nous passons la fonction `clicBouton` comme écouteur. Lorsque l'utilisateur clique sur le bouton, il diffuse un événement `click` qui est écouté par la fonction `clicBouton`.

---

Attention à ne pas mettre de doubles parenthèses lorsque nous passons une référence à la fonction écouteur. Nous devons passer sa référence et non le résultat de son exécution.

---

Revenons un instant sur notre code précédent :

```
monBouton.addEventListener ( "click", clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{
    // affiche : événement diffusé
    trace("événement diffusé");
}
```

Même si cette syntaxe fonctionne, et ne pose aucun problème pour le moment, que se passe t-il si nous spécifions un nom d'événement incorrect ?

Imaginons le cas suivant, nous faisons une erreur et écoutons l'événement `clieck` :

```
monBouton.addEventListener( "clieck", clicBouton );
```

Si nous testons le code ci-dessus, la fonction `clicBouton` ne sera jamais déclenchée car le nom de l'événement `click` possède une erreur de saisie et de par sa non-existence n'est jamais diffusé.

Afin d'éviter ces problèmes de saisie, tous les noms des événements sont désormais stockés dans des classes liées à chaque événement. Lorsque vous devez écouter un événement spécifique pensez immédiatement au type d'événement qui sera diffusé.

Les classes résidant dans le paquetage `flash.events` constituent l'ensemble des classes événementielles en ActionScript 3. Elles peuvent être divisées en deux catégories.

On peut ainsi distinguer les classes événementielles liées aux *événements autonomes* :

- `flash.events.Event`
- `flash.events.FullScreenEvent`
- `flash.events.HTTPStatusEvent`
- `flash.events.IOErrorEvent`
- `flash.events.NetStatusEvent`
- `flash.events.ProgressEvent`
- `flash.events.SecurityErrorEvent`
- `flash.events.StatusEvent`
- `flash.events.SyncEvent`
- `flash.events.TimerEvent`

Puis, celles liées aux *événements interactifs* :

- `flash.events.FocusEvent`
- `flash.events.IMEvent`
- `flash.events.KeyboardEvent`
- `flash.events.MouseEvent`
- `flash.events.TextEvent`

Dans notre cas, nous souhaitons écouter un événement qui relève d'une interactivité utilisateur provenant de la souris. Nous nous dirigerons logiquement au sein de la classe `MouseEvent`.

Pour récupérer le nom d'un événement, nous allons cibler ces propriétés constantes statiques, la syntaxe repose sur le système suivant :

```
| ClasseEvenement.MON_EVENEMENT
```

Pour bien comprendre ce concept, créez un nouveau document Flash et sur un calque AS tapez la ligne suivante :

```
| // affiche : click  
| trace ( MouseEvent.CLICK );
```

La classe `MouseEvent` contient tous les événements relatifs à la souris, en ciblant la propriété constante statique `CLICK` nous récupérerons la chaîne de caractères `click` que nous passerons en premier paramètre de la méthode `addEventListener`.

En ciblant un nom d'événement par l'intermédiaire d'une propriété de classe nous bénéficions de deux avantages. Si jamais une erreur de saisie survenait de par la vérification du code effectuée par le compilateur notre code ne pourrait pas être compilé.

Ainsi le code suivant échouerait à la compilation :

```
monBouton.addEventListener ( MouseEvent.CLICCK, clicBouton );
```

Il afficherait dans la fenêtre de sortie le message d'erreur suivant :

```
1119: Accès à la propriété CLICCK peut-être non définie, via la  
référence de type static Class.
```

Au sein de Flash CS3, Flash CS4 ou Flex Builder le simple fait de cibler une classe événementielle comme `MouseEvent` vous affiche aussitôt tout les événements liés à cette classe. Vous n'aurez plus aucune raison de vous tromper dans le ciblage de vos événements.

A plus long terme, si le nom de l'événement `click` venait à changer ou disparaître dans une future version du lecteur Flash, notre ancien code pointant vers la constante continuerait de fonctionner car ces propriétés seront mises à jour dans les futures versions du lecteur Flash.

Pour écouter le clic souris sur notre bouton, nous récupérerons le nom de l'événement et nous le passons à la méthode `addEventListener` :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );
```

Cette nouvelle manière de stocker le nom des événements apporte donc une garantie à la compilation mais règle aussi un souci de compatibilité future. Voyons ensemble les différentes classes liées aux événements existant en ActionScript 3.

## L'objet événementiel

Lorsqu'un événement est diffusé, un objet est obligatoirement envoyé en paramètre à la fonction écouteur. Cet objet est appelé *objet événementiel*.

Pour cette raison, la fonction écouteur doit impérativement contenir dans sa signature un paramètre du type de l'événement diffusé. Afin de ne jamais vous tromper, souvenez-vous de la règle suivante.

---

Le type de l'objet événementiel correspond toujours au type de la classe contenant le nom de l'événement.

---

Dans notre exemple il s'agit de `MouseEvent`. L'objet événementiel contient des informations liées à l'événement en cours de diffusion. Ces informations sont disponibles à travers des propriétés de l'objet événementiel.

Nous nous attarderons pour l'instant sur trois de ces propriétés, `target`, `currentTarget` et `type`.

- La propriété `target` fait référence à l'objet cible de l'événement. De n'importe où nous pouvons savoir qui est à l'origine de la propagation de l'événement.
- La propriété `currentTarget` fait référence à l'objet diffuseur de l'événement (le sujet). Il s'agit **toujours** de l'objet sur lequel nous avons appelé la méthode `addEventListener`.
- La propriété `type` contient, elle, le nom de l'événement diffusé, ici `click`.

Le code suivant récupère une référence vers l'objet cible, le sujet, ainsi que le nom de l'événement :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{
    // affiche : [object SimpleButton]
    trace( pEvt.target );

    // affiche : [object SimpleButton]
    trace( pEvt.currentTarget );

    // affiche : click
    trace( pEvt.type );
}
```

Nous reviendrons au cours du chapitre 6 intitulé *Propagation événementielle* sur les différences subtiles entre les propriétés `target` et `currentTarget`.

Le paramètre `pEvt` définit un paramètre de type `MouseEvent` car l'événement écouté est stocké au sein de la classe `MouseEvent`. Les propriétés `target` et `currentTarget`, présentent dans tout événement diffusé, permettent en réalité un couplage faible entre les objets. En passant par la propriété `currentTarget` pour cibler le sujet, nous évitons de le référencer directement par son nom, ce qui en cas de modification rendrait notre code rigide et pénible à modifier.

---

Attention, Si la fonction écouteur ne possède pas dans sa signature un paramètre censé accueillir l'objet événementiel, une exception est levée à l'exécution.

---

Si nous testons le code suivant :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ():void

{

}
```

L'erreur d'exécution suivante est levée :

```
ArgumentError: Error #1063: Non-correspondance du nombre d'arguments sur
bouton_fla::MainTimeline/onMouseClicked(). 0 prévu(s), 1 détecté(s).
```

Lorsque l'événement est diffusé, aucun paramètre n'accueille l'objet événementiel, une exception est donc levée. Souvenez-vous que la nouvelle machine virtuelle (AVM2) conserve les types à l'exécution, si le type de l'objet événementiel dans la signature de la fonction écouteur ne correspond pas au type de l'objet événementiel diffusé, le lecteur tentera de convertir implicitement l'objet événementiel. Une erreur sera aussitôt levée si la conversion est impossible.

Mettons en évidence ce comportement en spécifiant au sein de la signature de la fonction écouteur un objet événementiel de type différent.

Ici nous spécifions au sein de la signature de la fonction écouteur, un paramètre de type `flash.events.TextEvent` :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:TextEvent ):void

{

}
```

Ce code lève l'erreur d'exécution suivante :

```
TypeError: Error #1034: Echec de la contrainte de type : conversion de
flash.events::MouseEvent@2ee7601 en flash.events.TextEvent impossible.
```

A l'inverse, nous pouvons utiliser le type commun `flash.events.Event` compatible avec toutes les classes événementielles :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:Event ):void

{

}
```

```
| }  
| }
```

Nous pouvons nous demander l'intérêt d'une telle écriture. Pourquoi utiliser le type `Event` alors que nous attendons le type `MouseEvent` ?

Imaginons que nous souhaitions utiliser une seule et unique fonction écouteur pour de multiples événements. Prenons l'exemple d'une fonction `ecouteurClicGlobal` qui se charge d'écouter deux événements de types différents :

```
| monBouton.addEventListener ( MouseEvent.CLICK, ecouteurClicGlobal );  
| monChampTexte.addEventListener ( TextEvent.LINK, ecouteurClicGlobal );  
  
| function ecouteurClicGlobal ( pEvt:Event ):void  
| {  
| }  
| }
```

L'utilisation du type commun `Event` résout ici tout problème de contrainte de type. Nous reviendrons sur ce mécanisme au cours du chapitre 13 intitulé *Chargement de contenu*.

Si le code précédent vous pose un problème de compréhension, lisez avec attention les deux prochaines parties dédiées à la classe `Event`.

## La classe Event

Comme nous l'avons vu précédemment, au sein du paquetage `flash.events` réside un ensemble de classes contenant tous les types d'objets d'événementiels pouvant être diffusés.

---

Il est important de retenir que la classe `Event` est la classe parente de toutes les classes événementielles.

---

Cette classe définit la majorité des événements diffusés en ActionScript 3, le classique événement `Event.ENTER_FRAME` est contenu dans la classe `Event`, tout comme l'événement `Event.COMPLETE` indiquant la fin de chargement de données externe.

Dans un nouveau document Flash, créez un symbole `MovieClip` et placez une occurrence de ce clip nommé `monClip` sur la scène.

Le code suivant permet l'écoute de l'événement `Event.ENTER_FRAME` auprès de ce dernier :

```
| monClip.addEventListener ( Event.ENTER_FRAME, execute );  
  
| function execute ( pEvt:Event ):void  
| {  
| }
```



```
// affiche : [object MovieClip] : enterFrame
trace( pEvt.currentTarget + " : " + pEvt.type );

}
```

L'événement `Event.ENTER_FRAME` a la particularité d'être automatiquement diffusé dès lors que le lecteur entre sur une nouvelle image. Comme nous l'avons abordé précédemment, certains événements diffusent en revanche des objets événementiels de type étendus à la classe `Event`, c'est le cas notamment des événements provenant de la souris ou du clavier.

Lorsqu'un événement lié à la souris est diffusé, nous ne recevons pas un objet événementiel de type `Event` mais un objet de type `flash.events.MouseEvent`.

Pourquoi recevons-nous un objet événementiel de type différent pour les événements souris ?

C'est ce que nous allons découvrir ensemble dans cette partie intitulée *Les sous classes d'Event*.

## Les sous-classes d'Event

Lorsque l'événement `Event.ENTER_FRAME` est diffusé, l'objet événementiel de type `Event` n'a aucune information supplémentaire à renseigner à l'écouteur. Les propriétés `target` et `type` commune à ce type sont suffisantes.

A l'inverse, si vous souhaitez écouter un événement lié au clavier, il y a de fortes chances que notre écouteur ait besoin d'informations supplémentaires, comme par exemple la touche du clavier qui vient d'être enfoncée et qui a provoqué la diffusion de cet événement. De la même manière un événement diffusé par la souris pourra nous renseigner sur sa position ou bien sur quel élément notre curseur se situe.

Si nous regardons la définition de la classe `MouseEvent` nous découvrons de nouvelles propriétés contenant les informations dont nous pourrions avoir besoin.

Le code suivant récupère la position de la souris ainsi que l'état de la touche ALT du clavier :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{

    // affiche : x : 40.05 y : 124.45
```

```
    trace ( "x : " + pEvt.stageX + " y : " + pEvt.stageY );  
  
    // affiche : x : 3 y : 4  
    trace ( "x : " + pEvt.localX + " y : " + pEvt.localY );  
  
    // affiche : ALT enfoncé : false  
    trace ( "ALT enfoncé : " + pEvt.altKey );  
}
```

Lorsque l'événement `MouseEvent.CLICK` est diffusé, la fonction écouteur `clicBouton` en est notifiée et récupère les informations relatives à l'événement au sein de l'objet événementiel, ici de type `MouseEvent`.

Nous récupérons ici la position de la souris par rapport à la scène en ciblant les propriétés `stageX` et `stageY`, la propriété `altKey` est un booléen renseignant sur l'état de la touche ALT du clavier.

Les propriétés `localX` et `localY` de l'objet événementiel `pEvt` nous renseignent sur la position de la souris par rapport au repère local du bouton, nous disposons ici d'une très grande finesse en matière de gestion des coordonnées de la souris.

En examinant les autres sous-classes de la classe `Event` vous découvrirez que chacune des sous-classes en héritant intègre de nombreuses propriétés supplémentaires riches en informations.

---

## A retenir

---

- Tous les objets issus du paquetage `flash` peuvent diffuser des événements.
- La méthode `addEventListener` est le seul et unique moyen d'écouter un événement.
- Le nom de chaque événement est stocké dans une propriété statique de classe correspondant à l'événement en question.
- Les classes contenant le nom des événements sont appelées classes événementielles.
- La classe `Event` est la classe parente de toutes les classes événementielles.
- Lorsqu'un événement est diffusé, il envoie en paramètre à la fonction écouteur un objet appelé objet événementiel.
- Le type de cet objet événementiel est le même que la classe contenant le nom de l'événement.
- Un objet événementiel possède au minimum une propriété `target` et `currentTarget` référençant l'objet cible et l'objet auteur de l'événement. La propriété `type` permet de connaître le nom de l'événement en cours de diffusion.

### Arrêter l'écoute d'un événement

Imaginez que vous ne souhaitiez plus être mis au courant des dernières nouveautés DVD de votre vidéoclub. En informant le gérant que vous ne souhaitez plus en être notifié, vous ne serez plus écouteur de l'événement « nouveautés DVD ».

En ActionScript 3, lorsque nous souhaitons ne plus écouter un événement, nous utilisons la méthode `removeEventListener` dont voici la signature :

```
removeEventListener(type:String, listener:Function, useCapture:Boolean = false):void
```

Le premier paramètre appelé `type` attend le nom de l'événement auquel nous souhaitons nous désinscrire, le deuxième attend une référence à la fonction écouteur, le dernier paramètre sera traité au cours du chapitre 6 intitulé *Propagation événementielle*.

Reprenons ensemble notre exemple précédent. Lorsqu'un clic sur le bouton se produit, l'événement `MouseEvent.CLICK` est diffusé, la fonction écouteur `clicBouton` en est notifiée et nous supprimons l'écoute de l'événement :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{
```

```
    // affiche : x : 40.05 y : 124.45
    trace( "x : " + pEvt.stageX + " y : " + pEvt.stageY );

    // affiche : ALT enfoncé : false
    trace( "ALT enfoncé : " + pEvt.altKey );

    // on supprime l'écoute de l'événement MouseEvent.CLICK
    pEvt.target.removeEventListener ( MouseEvent.CLICK, clicBouton );
}
```

L'appel de la méthode `removeEventListener` sur l'objet `monBouton` au sein de la fonction écouteur `clicBouton`, supprime l'écoute de l'événement `MouseEvent.CLICK`. La fonction écouteur sera donc déclenchée une seule et unique fois.

Lorsque vous n'avez plus besoin d'un écouteur pensez à le supprimer de la liste des écouteurs à l'aide de la méthode `removeEventListener`. Le *ramasse-miettes* ne supprimera pas de la mémoire un objet ayant une de ses méthodes enregistrées comme écouteur. Pour optimiser votre application, pensez à supprimer les écouteurs non utilisés, l'occupation mémoire sera moindre et vous serez à l'abri de comportements difficiles à diagnostiquer.

## Mise en application

Afin de reprendre les notions abordées précédemment nous allons ensemble créer un projet dans lequel une fenêtre s'agrandit avec un effet d'inertie selon des dimensions évaluées aléatoirement. Une fois le mouvement terminé, nous supprimerons l'événement nécessaire afin de libérer les ressources et d'optimiser l'exécution de notre projet.

Dans un document Flash, nous créons une occurrence de clip appelée `myWindow` représentant un rectangle de couleur unie.

Nous écoutons l'événement `MouseEvent.CLICK` sur un bouton nommé `monBouton` placé lui aussi sur la scène :

```
| monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );
```

Puis nous définissons la fonction écouteur :

```
| function clicBouton ( pEvt:MouseEvent ):void
| {
|     trace("fonction écouteur déclenchée");
| }
```

Lors du clic souris sur le bouton `monBouton`, la fonction écouteur `clicBouton` est déclenchée, nous affichons un message validant

l'exécution de celle-ci. Il nous faut désormais écouter l'événement `Event.ENTER_FRAME` auprès de notre clip `myWindow`.

La fonction `clicBouton` est modifiée de la manière suivante :

```
function clicBouton ( pEvt:MouseEvent ):void
{
    trace("fonction écouteur déclenchée");
    trace("souscription de l'événement Event.ENTER_FRAME");
    myWindow.addEventListener ( Event.ENTER_FRAME, redimensionne );
}
```

En appelant la méthode `addEventListener` sur notre clip `myWindow`, nous souscrivons la fonction écouteur `redimensionne`, celle-ci s'occupera du redimensionnement de notre clip.

A la suite, définissons la fonction `redimensionne` :

```
function redimensionne ( pEvt:Event ):void
{
    trace("exécution de la fonction redimensionne");
}
```

Nous obtenons le code complet suivant :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{
    trace("fonction écouteur déclenchée");
    trace("souscription de l'événement Event.ENTER_FRAME");
    myWindow.addEventListener ( Event.ENTER_FRAME, redimensionne );
}

function redimensionne ( pEvt:Event ):void
{
    trace("exécution de la fonction redimensionne");
}
```

Au clic souris, le message "exécution de la fonction redimensionne" s'affiche. Ajoutons à présent un effet d'inertie pour gérer le redimensionnement de notre fenêtre.

Il nous faut dans un premier temps générer une dimension aléatoire. Pour cela nous définissons deux variables sur notre scénario, afin de stocker la largeur et la hauteur générées aléatoirement.

Juste après l'écoute de l'événement `MouseEvent.CLICK` nous définissons deux variables `largeur` et `hauteur` :

```
var largeur:int;  
var hauteur:int;
```

Ces deux variables de scénario serviront à stocker les tailles en largeur et hauteur que nous allons générer aléatoirement à chaque clic souris.

---

Notons que l'utilisation du type `int` pour les variables `hauteur` et `largeur` nous permet d'obtenir par défaut des valeurs arrondies à l'entier inférieur.

Cela nous évite de faire la conversion manuellement à l'aide de la méthode `floor` de la classe `Math`.

---

Nous évaluons ces dimensions au sein de la fonction `clicBouton` en affectant les deux variables :

```
function clicBouton ( pEvt:MouseEvent ):void  
{  
    largeur = Math.random()*400;  
    hauteur = Math.random()*400;  
  
    trace("fonction écouteur déclenchée");  
  
    trace("souscription de l'événement Event.ENTER_FRAME");  
  
    myWindow.addEventListener ( Event.ENTER_FRAME, redimensionne );  
}
```

Nous utilisons la méthode `random` de la classe `Math` afin d'évaluer une dimension aléatoire en largeur et hauteur dans une amplitude de 400 pixels. Nous choisissons ici une amplitude inférieure à la taille totale de la scène.

Nous modifions la fonction `redimensionne` afin d'ajouter l'effet voulu :

```
function redimensionne ( pEvt:Event ):void  
{  
    trace("exécution de la fonction redimensionne");  
  
    myWindow.width -= ( myWindow.width - largeur ) * .3;  
    myWindow.height -= ( myWindow.height - hauteur ) * .3;
```

```
}
```

Rafraîchissons-nous la mémoire et lisons le code complet :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton );

var largeur:int;
var hauteur:int;

function clicBouton ( pEvt:MouseEvent ):void
{
    largeur = Math.random()*400;
    hauteur = Math.random()*400;

    trace("fonction écouteur déclenchée");

    trace("souscription de l'événement Event.ENTER_FRAME");

    myWindow.addEventListener ( Event.ENTER_FRAME, redimensionne );
}

function redimensionne ( pEvt:Event ):void
{
    trace("exécution de la fonction redimensionne");

    myWindow.width -= ( myWindow.width - largeur ) *.3;
    myWindow.height -= ( myWindow.height - hauteur ) *.3;
}
```

Lorsque nous cliquons sur le bouton `monBouton`, notre fenêtre se redimensionne à une taille aléatoire, l'effet est visuellement très sympathique mais n'oublions pas que nous n'arrêtons jamais l'écoute de l'événement `Event.ENTER_FRAME` !

---

Il est important de noter qu'une fois un écouteur enregistré auprès d'un événement spécifique, toute nouvelle souscription est ignorée. Ceci évite qu'un écouteur ne soit enregistré plusieurs fois auprès d'un même événement.

Ainsi dans notre code, quelque soit le nombre de clic, la fonction `redimensionne` n'est souscrite qu'une seule fois à l'événement `Event.ENTER_FRAME`.

---

En laissant le code ainsi, même lorsque le mouvement de la fenêtre est terminé notre fonction écouteur `redimensionne` continue de s'exécuter et ralentit grandement notre application.

La question que nous devons nous poser est la suivante :

Quand devons-nous supprimer l'écoute de l'événement `Event.ENTER_FRAME` ?

L'astuce consiste à tester la différence entre la largeur et hauteur actuelle et la largeur et hauteur finale. Si la différence est inférieure à un demi nous pouvons en déduire que nous sommes quasiment arrivés à destination, et donc pouvons supprimer l'événement `Event.ENTER_FRAME`.

Afin d'exprimer cela dans notre code, rajoutez cette condition au sein de la fonction `redimensionne` :

```
function redimensionne ( pEvt:Event ):void
{
    trace("exécution de la fonction redimensionne");

    myWindow.width -= ( myWindow.width - largeur ) * .3;
    myWindow.height -= ( myWindow.height - hauteur ) * .3;

    if ( Math.abs ( myWindow.width - largeur ) < .5 && Math.abs (
myWindow.height - hauteur ) < .5 )
    {
        myWindow.removeEventListener ( Event.ENTER_FRAME, redimensionne );
        trace("redimensionnement terminé");
    }
}
```

Nous supprimons l'écoute de l'événement `Event.ENTER_FRAME` à l'aide de la méthode `removeEventListener`. La fonction `redimensionne` n'est plus exécutée, de cette manière nous optimisons la mémoire, et donc les performances d'exécution de notre application.

Notre application fonctionne sans problème, pourtant un point essentiel a été négligé ici, nous n'avons absolument pas tiré parti d'une fonctionnalité du nouveau modèle événementiel.

Notre *couplage* entre nos objets est dit *fort*, cela signifie qu'un changement de nom sur un objet entraînera de lourdes modifications en chaîne au sein de notre code. Voyons comment arranger cela en optimisant nos relations inter-objets.

---

## A retenir

---



- Lorsqu'un événement n'est plus utilisé, pensez à arrêter son écoute à l'aide de la méthode `removeEventListener`.

## La puissance du couplage faible

Notre code précédent fonctionne pour le moment sans problème. Si nous devons modifier certains éléments de notre application, comme par exemple le nom de certains objets, une modification en cascade du code serait inévitable.

Pour illustrer les faiblesses de notre code précédent, imaginons le scénario suivant :

Marc, un nouveau développeur ActionScript 3 intègre votre équipe et reprend le projet de redimensionnement de fenêtre que nous avons développé ensemble. Marc décide alors de renommer le clip `myWindow` par `maFenetre` pour des raisons pratiques. Comme tout bon développeur, celui-ci s'attend à mettre à jour une partie minime du code.

Marc remplace donc la ligne qui permet la souscription de l'événement `Event.ENTER_FRAME` au clip fenêtre.

Voici sa nouvelle version de la fonction `clicBouton` :

```
function clicBouton ( pEvt:MouseEvent ):void
{
    largeur = Math.random()*400;
    hauteur = Math.random()*400;

    trace("fonction écouteur déclenchée");

    trace("souscription de l'événement Event.ENTER_FRAME");

    maFenetre.addEventListener ( Event.ENTER_FRAME, redimensionne );
}
```

A la compilation, Marc se rend compte des multiples erreurs affichées dans la fenêtre de sortie, à plusieurs reprises l'erreur suivante est affichée :

```
1120: Accès à la propriété non définie myWindow.
```

Cette erreur signifie qu'une partie de notre code fait appel à cet objet `myWindow` qui n'est pourtant plus présent dans notre application. En effet, au sein de notre fonction `redimensionne` nous ciblons toujours l'occurrence `myWindow` qui n'existe plus car Marc l'a renommé.

Marc décide alors de mettre à jour la totalité du code de la fonction `redimensionne` et obtient le code suivant :

```
function redimensionne ( pEvt:Event ):void
{
    trace("exécution de la fonction redimensionne");

    maFenetre.width -= ( maFenetre.width - largeur ) * .3;
    maFenetre.height -= ( maFenetre.height - hauteur ) * .3;

    if ( Math.abs ( maFenetre.width - largeur ) < .5 && Math.abs (
maFenetre.height - hauteur ) < .5 )
    {
        maFenetre.removeEventListener ( Event.ENTER_FRAME, redimensionne );

        trace("arrivé");
    }
}
```

A la compilation, tout fonctionne à nouveau !

Malheureusement pour chaque modification du nom d'occurrence du clip `maFenetre`, nous devons mettre à jour la totalité du code de la fonction `redimensionne`. Cela est dû au fait que notre couplage inter-objets est fort.

Eric qui vient de lire un article sur le couplage faible propose alors d'utiliser au sein de la fonction `redimensionne` une information essentielle apportée par tout objet événementiel diffusé.

Souvenez-vous, lorsqu'un événement est diffusé, les fonctions écouteurs sont notifiées de l'événement, puis un objet événementiel est passé en paramètre à chaque fonction écouteur. Au sein de tout objet événementiel résident les propriétés `target` et `currentTarget` renseignant la fonction écouteur sur l'objet cible ainsi que l'auteur de l'événement. Grâce à la propriété `currentTarget` ou `target`, nous rendons le couplage *faible* entre nos objets.

Modifions le code de la fonction `redimensionne` de manière à cibler la propriété `currentTarget` de l'objet événementiel :

```
function redimensionne ( pEvt:Event ):void
{
    trace("exécution de la fonction redimensionne");

    var objetDiffuseur:DisplayObject = pEvt.currentTarget as DisplayObject;
```

```
objetDiffuseur.width -= ( objetDiffuseur.width - nLargeur ) * .3;
objetDiffuseur.height -= ( objetDiffuseur.height - nHauteur ) * .3;

if ( Math.abs ( objetDiffuseur.width - nLargeur ) < .5 && Math.abs (
objetDiffuseur.height - nHauteur ) < .5 )
{
    objetDiffuseur.removeEventListener ( Event.ENTER_FRAME, redimensionne
);
    trace("arrivé");
}
}
```

Si nous compilons, le code fonctionne. Désormais la fonction `redimensionne` fait référence au clip `maFenetre` par l'intermédiaire de la propriété `currentTarget` de l'objet événementiel.

---

Souvenez-vous, la propriété `currentTarget` fait toujours référence à l'objet sujet sur lequel nous avons appelé la méthode `addEventListener`.

---

Ainsi, lorsque le nom d'occurrence du clip `maFenetre` est modifié la propriété `currentTarget` nous permet de cibler l'objet auteur de l'événement sans même connaître son nom ni son emplacement.

Pour toute modification future du nom d'occurrence du clip `maFenetre`, aucune modification ne devra être apportée à la fonction écouteur `redimensionne`. Nous gagnons ainsi en souplesse, notre code est plus simple à maintenir.

Quelques jours plus tard, Marc décide d'imbriquer le clip `maFenetre` dans un autre clip appelé `conteneur`.

Heureusement, notre couplage inter-objets est faible, une seule ligne seulement devra être modifiée :

```
function clicBouton ( pEvt:MouseEvent ):void
{
    largeur = Math.random()*400;
    hauteur = Math.random()*400;

    trace("fonction écouteur déclenchée");

    trace("souscription de l'événement Event.ENTER_FRAME");

    conteneur.maFenetre.addEventListener ( Event.ENTER_FRAME, redimensionne );
}
```

Marc est tranquille, grâce au couplage faible son travail de mise à jour du code est quasi nul.

## A retenir

- Lorsqu'un événement n'a plus besoin d'être observé, nous supprimons l'écoute avec la méthode `removeEventListener`.
- La méthode `removeEventListener` est le seul et unique moyen pour supprimer l'écoute d'un événement.
- Le nom de chaque événement est stocké dans une propriété statique de classe correspondant à l'événement en question.
- Utilisez la propriété `target` ou `currentTarget` des objets événementiels pour garantir un couplage faible entre les objets.
- Nous découvrirons au cours du chapitre 6 intitulé *Propagation événementielle*, les différences subtiles entre les propriétés `target` et `currentTarget`.

## Souplesse de code

L'intérêt du nouveau modèle événementiel ActionScript 3 ne s'arrête pas là. Auparavant il était impossible d'affecter plusieurs fonctions écouteurs à un même événement.

Le code ActionScript suivant met en évidence cette limitation de l'ancien modèle événementiel :

```
function clicBouton1 ( )
{
    trace("click 1 sur le bouton");
}

function clicBouton2 ( )
{
    trace("click 2 sur le bouton");
}

monBouton.onRelease = clicBouton1;
monBouton.onRelease = clicBouton2;
```

Nous définissions sur le bouton `monBouton` deux fonctions pour gérer l'événement `onRelease`. En utilisant ce modèle événementiel il nous était impossible de définir plusieurs fonctions pour gérer un même événement.

---

En ActionScript 1 et 2, une seule et unique fonction pouvait être définie pour gérer un événement spécifique.

---

Dans notre code, seule la fonction `clicBouton2` était affectée comme gestionnaire de l'événement `onRelease`, car la dernière affectation effectuée sur l'événement `onRelease` est la fonction `clicBouton2`. La deuxième affectation écrasait la précédente.

Le principe même du nouveau modèle événementiel apporté par ActionScript 3 repose sur le principe d'une relation un à plusieurs, c'est la définition même du modèle de conception *Observateur*.

De ce fait nous pouvons souscrire plusieurs écouteurs auprès du même événement. Prenons un exemple simple constitué d'un bouton et de deux fonctions écouteurs.

Dans un nouveau document Flash, créez un symbole bouton et placez une occurrence de bouton sur la scène et nommez la `monBouton`.

Sur un calque AS tapez le code suivant :

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton1 );
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton2 );

function clicBouton1 ( pEvt:MouseEvent ):void
{
    // affiche : clicBouton1 : [object MovieClip] : monBouton
    trace( "clicBouton1 : " + pEvt.currentTarget + " : " +
    pEvt.currentTarget.name );
}

function clicBouton2 ( pEvt:MouseEvent ):void
{
    // affiche : clicBouton2 : [object MovieClip] : monBouton
    trace( "clicBouton2 : " + pEvt.currentTarget + " : " +
    pEvt.currentTarget.name );
}
```

Nous souscrivons deux fonctions écouteurs auprès de l'événement `MouseEvent.CLICK`, au clic bouton les deux fonctions écouteurs sont notifiées de l'événement.

Sachez que l'ordre de notification dépend de l'ordre dans lequel les écouteurs ont été souscrits. Ainsi dans notre exemple la fonction `clicBouton2` sera exécutée après la fonction `clicBouton1`.

De la même manière, une seule fonction écouteur peut être réutilisée pour différents objets. Dans le code suivant la fonction écouteur `tourner` est utilisée pour les trois boutons.

Un premier réflexe pourrait nous laisser écrire le code suivant :

```
monBouton1.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );
monBouton2.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );
monBouton3.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );

function tourner ( pEvt:MouseEvent )
{
    if ( pEvt.currentTarget == monBouton1 ) monBouton1.rotation += 5;

    else if ( pEvt.currentTarget == monBouton2 ) monBouton2.rotation += 5;

    else if ( pEvt.currentTarget == monBouton3 ) monBouton3.rotation += 5;
}
```

Souvenez-vous que la propriété `currentTarget` vous permet de référencer dynamiquement l'objet auteur de l'événement :

```
monBouton1.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );
monBouton2.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );
monBouton3.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );

function tourner ( pEvt:MouseEvent )
{
    pEvt.currentTarget.rotation += 5;
}
```

En modifiant le nom des boutons, la fonction `tourner` ne subit, elle, aucune modification :

```
boutonA.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );
boutonB.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );
boutonC.addEventListener ( MouseEvent.MOUSE_DOWN, tourner );
```

La fonction `tourner` est ainsi réutilisable pour n'importe quel objet pouvant subir une rotation.

## Ordre de notification

Imaginez que vous soyez abonné à un magazine, vous souhaitez alors recevoir le magazine en premier lors de sa sortie, puis une fois reçu, le magazine est alors distribué aux autres abonnés.

Certes ce scénario est peu probable dans la vie mais il illustre bien le concept de priorité de notifications du modèle événementiel ActionScript 3.

Afin de spécifier un ordre de notification précis nous pouvons utiliser le paramètre `priority` de la méthode `addEventListener` dont nous revoyons la signature :

```
addEventListener(type:String, listener:Function, useCapture:Boolean = false,
priority:int = 0, useWeakReference:Boolean = false):void
```

En reprenant notre exemple précédent, nous spécifions le paramètre `priority`:

```
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton1, false, 0);
monBouton.addEventListener ( MouseEvent.CLICK, clicBouton2, false, 1);

function clicBouton1 ( pEvt:MouseEvent ):void
{
    // affiche : A [object SimpleButton] : monBouton
    trace( "A " + pEvt.currentTarget + " : " + pEvt.currentTarget.name );
}

function clicBouton2 ( pEvt:MouseEvent ):void
{
    // affiche : B [object SimpleButton] : monBouton
    trace( "B " + pEvt.currentTarget + " : " + pEvt.currentTarget.name );
}
```

Lors du clic sur le bouton `monBouton`, la fonction `clicBouton1` est déclenchée après la fonction `clicBouton2`, le message suivant est affiché :

```
B [object SimpleButton] : monBouton
A [object SimpleButton] : monBouton
```

Une fonction écouteur enregistrée avec une priorité de 0 sera exécutée après une fonction écouteur enregistrée avec une priorité de 1.

Il est impossible de modifier la priorité d'un écouteur une fois enregistré, nous serions obligés de supprimer l'écouteur à l'aide de la méthode `removeEventListener`, puis d'enregistrer à nouveau l'écouteur.

---

Cette notion de priorité n'existe pas dans l'implémentation officielle DOM3 et a été rajoutée uniquement en ActionScript 3 pour offrir plus de flexibilité.

---

Même si cette fonctionnalité peut s'avérer pratique dans certains cas précis, s'appuyer sur l'ordre de notification dans vos développements est considéré comme une mauvaise pratique, la mise à jour de votre code peut s'avérer difficile, et rendre le déroulement de l'application complexe.

## Références faibles

En ActionScript 3 la gestion de la mémoire est assurée par une partie du lecteur appelée *ramasse-miettes* ou *Garbage Collector* en anglais. Nous avons rapidement abordé cette partie du lecteur au cours du chapitre 2 intitulé *Langage et API*.

Le ramasse-miettes est chargé de libérer les ressources en supprimant de la mémoire les objets qui ne sont plus utilisés. Un objet est considéré comme non utilisé lorsqu'aucune référence ne pointe vers lui, autrement dit lorsque l'objet est devenu inaccessible.

Quand nous écoutons un événement spécifique, l'écouteur est ajouté à la liste interne du diffuseur, il s'agit en réalité d'un tableau stockant les références de chaque écouteur.

---

Gardez à l'esprit que le sujet référence l'écouteur et non l'inverse.

---

De ce fait le ramasse-miettes ne supprimera jamais ces écouteurs tant que ces derniers seront *référéncés* par les sujets et que ces derniers demeurent référencés.

Lors de la souscription d'un écouteur auprès d'un événement vous avez la possibilité de modifier ce comportement grâce au dernier paramètre de la méthode `addEventListener`.

Revenons sur la signature de cette méthode :

```
addEventListener(type:String, listener:Function, useCapture:Boolean = false,
priority:int = 0, useWeakReference:Boolean = false):void
```

Le cinquième paramètre appelé `useWeakReference` permet de spécifier si la fonction écouteur sera stockée à l'aide d'une référence forte ou faible. En utilisant une référence faible, la fonction écouteur est référencée faiblement au sein du tableau d'écouteurs interne.

De cette manière, si la seule ou dernière référence à la fonction écouteur est une référence faible lors du passage du ramasse-miettes, ce dernier fait exception, ignore cette référence et supprime tout de même l'écouteur de la mémoire.

Attention, cela ne veut pas dire que la fonction écouteur va obligatoirement être supprimée de la mémoire. Elle le sera uniquement si le ramasse-miettes intervient et procède à un nettoyage.

---

Notez bien que cette intervention pourrait ne jamais avoir lieu si le lecteur Flash n'en décidait pas ainsi.

---



C'est pour cette raison qu'il ne faut pas considérer cette technique comme une suppression automatique des fonctions écouteurs dès lors qu'elles ne sont plus utiles. Pensez à toujours appeler la méthode `removeEventListener` pour supprimer l'écoute de certains événements lorsque cela n'est plus nécessaire.

Si ne nous spécifions pas ce paramètre, sa valeur par défaut est à `false`. Ce qui signifie que tous nos objets sujets conservent jusqu'à l'appel de la méthode `removeEventListener` une référence forte vers l'écouteur.

Dans le code suivant, une méthode d'instance de classe personnalisée est enregistrée comme écouteur de l'événement `Event.ENTER_FRAME` d'un `MovieClip` :

```
// instantiation d'un objet personnalisé
var monObjetPerso:ClassePerso = new ClassePerso();

var monClip:MovieClip = new MovieClip();

// écoute de l'événement Event.ENTER_FRAME
monClip.addEventListener( Event.ENTER_FRAME, monObjetPerso.ecouteur );

// supprime la référence vers l'instance de ClassePerso
monObjetPerso = null;
```

Bien que nous ayons supprimé la référence à l'instance de `ClassePerso`, celle-ci demeure référencée fortement par l'objet sujet `monClip`.

En s'inscrivant comme écouteur à l'aide d'une référence faible, le ramasse-miettes ignorera la référence détenue par le sujet et supprimera l'instance de `ClassePerso` de la mémoire :

```
// écoute de l'événement Event.ENTER_FRAME
monClip.addEventListener( Event.ENTER_FRAME, monObjetPerso.ecouteur, false,
0, true );
```

Les références faibles ont un intérêt majeur lorsque l'écouteur ne connaît pas l'objet sujet auquel il est souscrit. N'ayant aucun moyen de se désinscrire à l'événement, il est recommandé d'utiliser une référence faible afin de garantir que l'objet puisse tout de même être libéré de la mémoire.

Nous reviendrons sur ces subtilités tout au long de l'ouvrage afin de ne pas être surpris par le ramasse-miettes.

---

## A retenir

- En ajoutant un écouteur à l'aide d'une référence faible, ce dernier sera tout de même supprimé de la mémoire, même si ce dernier n'a

pas été désinscrit à l'aide de la méthode `removeEventListener`.

- Il ne faut surtout pas considérer l'utilisation de références faibles comme remplacement de la méthode `removeEventListener`.
- Veillez à bien désinscrire tous vos écouteurs lorsqu'ils ne sont plus utiles à l'aide de la méthode `removeEventListener`.

## Subtilités

Comme nous l'avons vu au début de ce chapitre, en ActionScript 1 et 2 nous avions l'habitude d'ajouter des écouteurs qui n'étaient pas des fonctions mais des objets. En réalité, une méthode portant le nom de l'événement était déclenchée lorsque ce dernier était diffusé.

Pour écouter un événement lié au clavier nous pouvions écrire le code suivant :

```
var monEcouteur = new Object();  
monEcouteur.onKeyDown = function ( )  
{  
    trace ( Key.getCode() );  
}  
Key.addListener ( monEcouteur );
```

En passant un objet comme écouteur, nous définissions une méthode portant le nom de l'événement diffusé, le lecteur déclenchait alors la méthode lorsque l'événement était diffusé. Désormais seule une fonction ou une méthode peut être passée en tant qu'écouteur.

Si nous testons le code suivant :

```
var monEcouteur:Object = new Object();  
stage.addEventListener ( KeyboardEvent.KEY_DOWN, monEcouteur );
```

L'erreur de compilation ci-dessous s'affiche dans la fenêtre de sortie :

```
1118: Contrainte implicite d'une valeur du type statique Object vers un type  
peut-être sans rapport Function.
```

Le compilateur détecte que le type de l'objet passé en tant qu'écouteur n'est pas une fonction mais un objet et refuse la compilation. En revanche il est techniquement possible de passer en tant qu'écouteur non pas une fonction mais une méthode créée dynamiquement sur un objet.

Le code suivant fonctionne sans problème :

```
var monEcouteur:Object = new Object();

monEcouteur.maMethode = function ()

{

    trace( "touche clavier enfoncée" );

}

stage.addEventListener ( KeyboardEvent.KEY_DOWN, monEcouteur.maMethode );
```

En revanche, une subtilité est à noter dans ce cas précis le contexte d'exécution de la méthode `maMethode` n'est pas celui que nous attendons. En faisant appel au mot-clé `this` pour afficher le contexte en cours nous serons surpris de ne pas obtenir `[object Object]` mais `[object global]`.

Lorsque nous passons une méthode stockée au sein d'une propriété dynamique d'un objet, le lecteur Flash n'a aucun moyen de rattacher cette méthode à son objet d'origine, la fonction s'exécute alors dans un contexte global, l'objet `global`. Le code suivant met en avant cette subtilité de contexte :

```
var monEcouteur:Object = new Object();

monEcouteur.maMethode = function ()

{

    // affiche : [object global]
    trace( this );

}

stage.addEventListener ( KeyboardEvent.KEY_DOWN, monEcouteur.maMethode );
```

Il est donc fortement déconseillé d'utiliser cette technique pour gérer les événements au sein de votre application.

Nous avons découvert ensemble le nouveau modèle événementiel, afin de couvrir totalement les mécanismes liés à ce modèle attardons-nous à présent sur la gestion de l'affichage en ActionScript 3.

## A retenir

- En ActionScript 3, un objet ne peut pas être utilisé comme écouteur.
- Si la méthode passée comme écouteur est créée dynamiquement, celle-ci s'exécute dans un contexte global.

Nous allons nous intéresser dans le chapitre suivant à la notion de liste d'affichage. Découvrons ensemble comment manipuler avec souplesse les objets graphiques au sein du lecteur Flash 9.

# 4

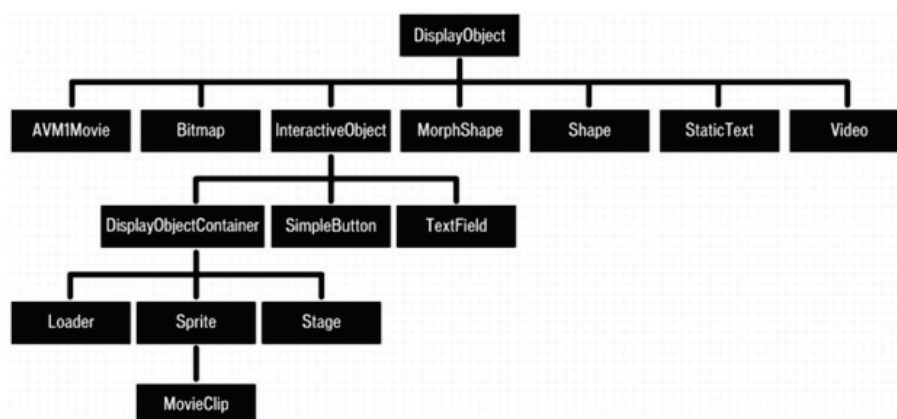
## La liste d’affichage

<b>CLASSES GRAPHIQUES .....</b>	<b>1</b>
<b>LISTE D’AFFICHAGE.....</b>	<b>5</b>
INSTANCIATION DES CLASSES GRAPHIQUES.....	7
AJOUT D’OBJETS D’AFFICHAGE .....	8
REATTRIBUTION DE CONTENEUR.....	9
LISTE INTERNE D’OBJETS ENFANTS .....	11
ACCEDER AUX OBJETS D’ AFFICHAGE .....	13
SUPPRESSION D’OBJETS D’ AFFICHAGE .....	20
EFFONDREMENT DES PROFONDEURS .....	25
GESTION DE L’EMPILEMENT DES OBJETS D’AFFICHAGE.....	27
ECHANGE DE PROFONDEURS .....	30
DESACTIVATION DES OBJETS GRAPHIQUES.....	32
FONCTIONNEMENT DE LA TETE DE LECTURE .....	35
SUBTILITES DE LA PROPRIETE STAGE.....	36
SUBTILITES DE LA PROPRIETE ROOT .....	38
LES _LEVEL.....	40

### Classes graphiques

En ActionScript 3 comme nous l’avons vu précédemment, toutes les classes graphiques sont stockées dans des emplacements spécifiques et résident au sein des trois différents paquetages : `flash.display`, `flash.media`, et `flash.text`.

Le schéma suivant regroupe l’ensemble d’entre elles :



*Figure 3-1. Classes graphiques de la liste d’affichage.*

Nous remarquons à travers ce schéma la multitude de classes graphiques disponibles pour un développeur ActionScript 3. La question que nous pouvons nous poser est la suivante.

Pourquoi un tel éclatement de classes ?

ActionScript 3 a été développé dans un souci d’optimisation. En offrant un large choix de classes graphiques nous avons la possibilité d’utiliser l’objet le plus optimisé pour chaque besoin. Plus la classe graphique est enrichie plus celle-ci occupe un poids en mémoire important. Du fait du petit nombre de classes graphiques en ActionScript 1 et 2 beaucoup de développeurs utilisaient la classe `MovieClip` pour tout réaliser, ce qui n’était pas optimisé.

---

En ActionScript 1 et 2 seules quatre classes graphiques existaient : `MovieClip`, `Button`, `TextField`, et dernièrement `BitmapData`.

---

Prenons l’exemple d’une application Flash traditionnelle comme une application de dessin. Avant ActionScript 3 le seul objet nous permettant de dessiner grâce à l’API de dessin était la classe `MovieClip`. Cette dernière intègre un scénario ainsi que des méthodes liées à la manipulation de la tête de lecture qui nous étaient inutiles dans ce cas précis. Un simple objet `Shape` plus léger en mémoire suffirait pour dessiner. Dans la même logique, nous préférons utiliser un objet `SimpleButton` plutôt qu’un `MovieClip` pour la création de boutons.

Nous devons mémoriser trois types d’objets graphiques primordiaux.

- Les objets de type `flash.display.DisplayObject`

Tout élément devant être affiché doit être de type `DisplayObject`. Un champ texte, un bouton ou un clip sera forcément de type

`DisplayObject`. Durant vos développements ActionScript 3, vous qualifieriez généralement de `DisplayObject` tout objet pouvant être affiché. La classe `DisplayObject` définit toutes les propriétés de base liées à l’affichage, la position, la rotation, l’étirement et d’autres propriétés plus avancées.

---

Attention : un objet héritant simplement de `DisplayObject` comme `Shape` par exemple n’a pas la capacité de contenir des objets graphiques. Parmi les sous-classes directes de `DisplayObject` nous pouvons citer les classes `Shape`, `Video`, `Bitmap`.

---

- Les objets de type `flash.display.InteractiveObject`

La classe `InteractiveObject` définit les comportements liés à l’interactivité. Lorsqu’un objet hérite de la classe `InteractiveObject`, ce dernier peut réagir aux entrées utilisateur liées à la souris ou au clavier. Parmi les objets graphiques descendant directement de la classe `InteractiveObject` nous pouvons citer les classes `SimpleButton` et `TextField`.

- Les objets de type `flash.display.DisplayObjectContainer`

A la différence des `DisplayObject` les `DisplayObjectContainer` peuvent contenir des objets graphiques. Dorénavant nous parlerons d’objet enfant pour qualifier un objet graphique contenu dans un autre. Les objets héritant de cette classe sont donc appelés *conteneurs d’objets d’affichage*. La classe `DisplayObjectContainer` est une sous classe de la classe `DisplayObject` et définit toutes les propriétés et méthodes relatives à la manipulation des objets enfants. `Loader`, `Sprite` et `Stage` héritent directement de `DisplayObjectContainer`.

Durant nos développements ActionScript 3, certains objets graphiques ne pourront être instanciés tandis que d’autres le pourront. Nous pouvons séparer en deux catégories l’ensemble des classes graphiques :

#### *Les objets instanciables :*

- `flash.display.Bitmap` : cette classe sert d’enveloppe à l’objet `flash.display.BitmapData` qui ne peut être ajouté à la liste d’affichage sans enveloppe `Bitmap`.
- `flash.display.Shape` : il s’agit de la classe de base pour tout contenu vectoriel, elle est fortement liée à l’API de dessin grâce à sa propriété `graphics`. Nous reviendrons sur l’API de dessin très vite.

- `flash.media.Video` : la classe `Video` sert à afficher les flux vidéo, provenant d’une webcam, d’un serveur de streaming, ou d’un simple fichier vidéo (FLV, MP4, MOV, etc.).
- `flash.text.TextField` : la gestion du texte est assurée par la classe `TextField`.
- `flash.display.SimpleButton` : la classe `SimpleButton` permet de créer des boutons dynamiquement ce qui était impossible dans les précédentes versions d’ActionScript.
- `flash.display.Loader` : la classe `Loader` gère le chargement de tout contenu graphique externe, chargement de SWF et images (PNG, GIF, JPG)
- `flash.display.Sprite` : la classe `Sprite` est un `MovieClip` allégé car il ne dispose pas de scénario.
- `flash.display.MovieClip` : la classe `MovieClip` est la classe graphique la plus enrichie, elle est liée à l’animation traditionnelle.

*Les objets non instanciables:*

- `flash.display.AVM1Movie` : lorsqu’une animation ActionScript 1 ou 2 est chargée au sein d’une animation ActionScript 3, le lecteur Flash 9 enveloppe l’animation d’ancienne génération dans un objet de type `AVM1Movie` il est donc impossible d’instancier un objet de ce type par programmation.
- `flash.display.InteractiveObject` : la classe `InteractiveObject` définit les comportements liés à l’interactivité comme les événements souris ou clavier par exemple.
- `flash.display.MorphShape` : les formes interpolées sont représentées par le type `MorphShape`. Seul l’environnement auteur de Flash CS3 peut créer des objets de ce type.
- `flash.display.StaticText` : la classe `StaticText` représente les champs texte statique créés dans l’environnement auteur Flash CS3
- `flash.display.Stage` : il est le conteneur principal de tout notre contenu graphique.

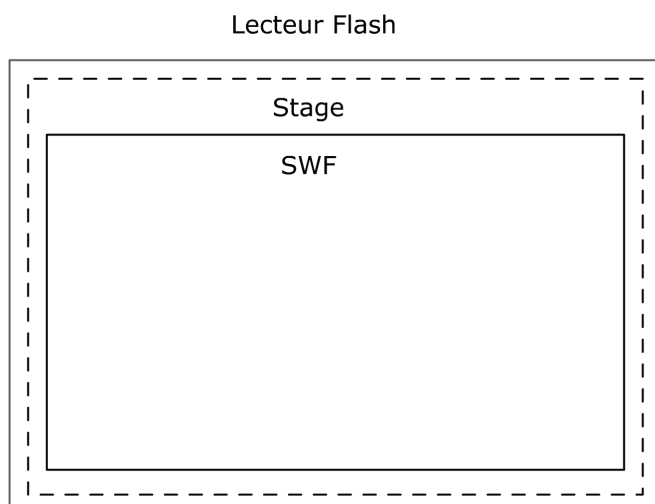
**A retenir**

- Il faut utiliser la classe la plus optimisée pour chaque cas.
- Les classes graphiques résident dans trois paquetages différents : `flash.display`, `flash.media`, et `flash.text`.
- Les classes `DisplayObject`, `InteractiveObject`, et `DisplayObjectContainer` sont abstraites et ne peuvent être instanciées ni héritées en ActionScript.

## Liste d’affichage

Lorsqu’une animation est chargée au sein du lecteur Flash, celui-ci ajoute automatiquement la scène principale du SWF en tant que premier enfant du conteneur principal, l’objet `Stage`. Cette hiérarchie définit ce qu’on appelle *la liste d’affichage* de l’application.

La figure 3-2 illustre le mécanisme :



*Figure 3-2. Schéma du système d’affichage du lecteur Flash.*

Afin de bien comprendre comment la liste d’affichage s’organise, créez un nouveau document Flash CS3 et testez le code suivant sur le scénario principal :

```
// affiche : [object MainTimeline]
trace( this );

// affiche : [object Stage]
trace( this.parent );
```

En faisant appel au mot-clé `this`, nous faisons référence à notre scénario principal l’objet `MainTimeline` contenu par l’objet `Stage`.



Attention, l’objet `Stage` n’est plus accessible de manière globale comme c’était le cas en ActionScript 1 et 2.

Le code suivant génère une erreur à la compilation :

```
| trace( Stage );
```

Pour faire référence à l’objet `Stage` nous devons obligatoirement passer par la propriété `stage` d’un `DisplayObject` présent au sein de la liste d’affichage :

```
| monDisplayObject.stage
```

Pour récupérer le nombre d’objets d’affichage contenu dans un objet de type `DisplayObjectContainer` nous utilisons la propriété `numChildren`.

Ainsi pour récupérer le nombre d’objets d’affichage contenus par l’objet `Stage` nous écrivons :

```
| // affiche : 1  
| trace( this.stage.numChildren );
```

Ou bien de manière implicite :

```
| // affiche : 1  
| trace( stage.numChildren );
```

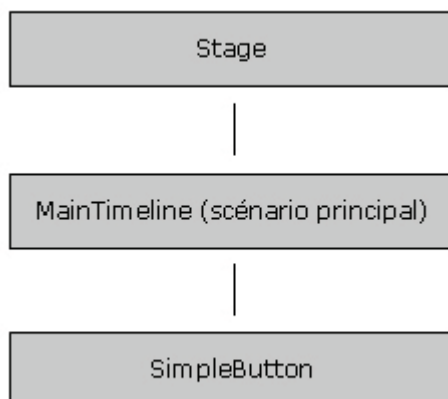
Nous reviendrons très bientôt sur l’accès à cette propriété qui mérite une attention toute particulière.

Notre objet `Stage` contient un seul enfant, notre scénario principal, l’objet `MainTimeline`. Ainsi lorsqu’un SWF vide est lu, deux `DisplayObject` sont présents par défaut dans la liste d’affichage :

- L’objet `Stage`
- Le scénario du SWF, l’objet `MainTimeline`

Comme l’illustre la figure 3-2, chaque application ActionScript 3 possède un seul objet `Stage` et donc une seule et unique liste d’affichage. Il faut considérer l’objet `Stage` comme le nœud principal de notre liste d’affichage. En réalité celle-ci peut être représentée comme une arborescence XML, un nœud principal duquel découlent d’autres nœuds enfants, nos objets d’affichages.

Lorsque nous posons une occurrence de bouton sur la scène principale de notre animation, nous obtenons la liste d’affichage suivante :



*Figure 3-3. Liste d’affichage avec un bouton.*

Nous verrons au cours du chapitre 11 intitulé *Classe du document* comment remplacer la classe `MainTimeline` par une sous classe graphique personnalisée.

Jusqu’à maintenant nous n’avons pas créé d’objet graphique par programmation. `ActionScript 3` intègre un nouveau procédé d’instanciation des classes graphiques bien plus efficace que nous allons aborder à présent.

## A retenir

- L’objet `Stage` n’est plus accessible de manière globale.
- Nous accédons à l’objet `Stage` à travers la propriété `stage` de tout `DisplayObject`.

## Instanciation des classes graphiques

Une des grandes nouveautés d’`ActionScript 3` concerne le procédé d’instanciation des classes graphiques. Avant `ActionScript 3`, nous devions utiliser diverses méthodes telles `createEmptyMovieClip` pour la création de `flash.display.MovieClip`, ou encore `createTextField` pour la classe `flash.text.TextField`.

Il nous fallait mémoriser plusieurs méthodes ce qui n’était pas forcément évident pour une personne découvrant `ActionScript`. Désormais, nous utilisons le mot clé `new` pour instancier tout objet graphique.

Le code suivant crée une nouvelle instance de `MovieClip`.

```
var monClip:MovieClip = new MovieClip();
```

De la même manière, si nous souhaitons créer un champ texte dynamiquement nous écrivons :

```
| var monChampTexte:TextField = new TextField();
```

Pour affecter du texte à notre champ nous utilisons la propriété `text` de la classe `TextField` :

```
| var monChampTexte:TextField = new TextField();  
| monChampTexte.text = "Hello World";
```

L’utilisation des méthodes `createTextField` ou `createEmptyMovieClip` nous obligeaient à conserver une référence à un `MovieClip` afin de pouvoir instancier nos objets graphiques, avec l’utilisation du mot-clé `new` tout cela n’est qu’un mauvais souvenir.

Nous n’avons plus besoin de spécifier de nom d’occurrence ni de profondeur lors de la phase d’instanciation. Bien que notre objet graphique soit créé et réside en mémoire, celui-ci n’a pas encore été ajouté à la liste d’affichage. Si nous testons le code précédent, nous remarquons que le texte n’est pas affiché.

Un des comportements les plus troublants lors de la découverte d’ActionScript 3 concerne les *objets d’affichage hors liste*.

## A retenir

- Pour qu’un objet graphique soit visible, ce dernier doit obligatoirement être ajouté à la liste d’affichage.
- Tous les objets graphiques s’instancient avec le mot clé `new`.

## Ajout d’objets d’affichage

En ActionScript 3, lorsqu’un objet graphique est créé, celui-ci n’est pas automatiquement ajouté à la liste d’affichage comme c’était le cas en ActionScript 1 et 2. L’objet existe en mémoire mais ne réside pas dans la liste d’affichage et n’est donc pas rendu.

Dans ce cas l’objet est appelé *objet d’affichage hors liste*.

Pour l’afficher nous utilisons une des deux méthodes définies par la classe `DisplayObjectContainer` appelées `addChild` et `addChildAt`.

Voici la signature de la méthode `addChild` :

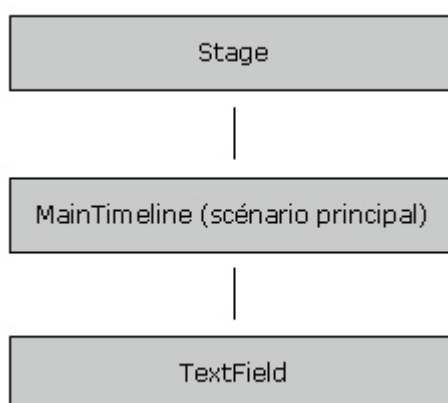
```
| public function addChild(child:DisplayObject):DisplayObject
```

La méthode `addChild` accepte comme seul paramètre une instance de `DisplayObject`. Pour voir notre objet graphique nous devons l’ajouter à la liste d’objets enfants d’un `DisplayObjectContainer` présent au sein de la liste d’affichage.

Nous pouvons ajouter par exemple l’instance à notre scénario principal :

```
var monChampTexte:TextField = new TextField();  
  
monChampTexte.text = "Hello World";  
  
addChild ( monChampTexte );
```

Nous obtenons la liste d’affichage illustrée par la figure suivante :



*Figure 3-4. Liste d’affichage simple avec un champ texte.*

Ce comportement nouveau est extrêmement puissant. Tout développeur ActionScript a déjà souhaité charger une image ou attacher un objet de la bibliothèque sans pour autant l’afficher.

Avec le concept d’objets d’affichage hors liste, un objet graphique peut « vivre » sans pour autant être rendu à l’affichage. Ce comportement offre de nouvelles possibilités en matière d’optimisation de l’affichage. Nous reviendrons sur ce mécanisme lors du chapitre 12 intitulé *Programmation bitmap*.

## Réattribution de conteneur

En ActionScript 1 et 2, aucune méthode n’existait pour déplacer un objet graphique au sein de la liste d’affichage. Il était impossible de changer son parent. Le seul moyen était de supprimer l’objet graphique puis le recréer au sein du conteneur voulu.

En ActionScript 3, si nous passons à la méthode `addChild` un `DisplayObject` déjà présent au sein de la liste d’affichage, ce dernier est supprimé de son conteneur d’origine et placé dans le nouveau `DisplayObjectContainer` sur lequel nous avons appelé la méthode `addChild`.

Prenons un exemple simple, nous créons un clip que nous ajoutons à la liste d’affichage en l’ajoutant à notre scénario principal :

```
var monClip:MovieClip = new MovieClip();  
  
// le clip monClip est ajouté au scénario principal  
addChild ( monClip );  
  
// affiche : 1  
trace( numChildren );
```

La propriété `numChildren` du scénario principal renvoie 1 car le clip `monClip` est un enfant du scénario principal.

Nous créons maintenant, un objet graphique de type `Sprite` et nous lui ajoutons en tant qu’enfant notre clip `monClip` déjà contenu par notre scénario principal :

```
var monClip:MovieClip = new MovieClip();  
  
// le clip monClip est ajouté au scénario principal  
addChild ( monClip );  
  
// affiche : 1  
trace( numChildren );  
  
var monSprite:Sprite = new Sprite();  
  
addChild ( monSprite );  
  
// affiche : 2  
trace( numChildren );  
  
monSprite.addChild ( monClip );  
  
// affiche : 1  
trace( numChildren );
```

La propriété `numChildren` du scénario principal renvoie toujours 1 car le clip `monClip` a quitté le scénario principal afin d’être placé au sein de notre objet `monSprite`.

Dans le code suivant, l’appel successif de la méthode `addChild` ne duplique donc pas l’objet graphique mais le déplace de son conteneur d’origine pour le replacer à nouveau :

```
var monClip:MovieClip = new MovieClip();  
  
// le clip monClip est ajouté au scénario principal  
addChild ( monClip );
```

```
// le clip monClip quitte le scénario principal
// puis est remplacé au sein de ce dernier
addChild ( monClip );

// affiche : 1
trace( numChildren );
```

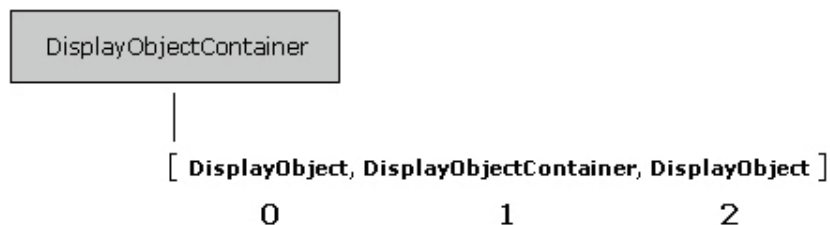
Il n’existe pas en ActionScript 3 d’équivalent à la méthode `duplicateMovieClip` existante dans les précédentes versions d’ActionScript. Pour dupliquer un objet graphique, nous utilisons le mot-clé `new`.

La réattribution de conteneurs illustre la puissance apportée par le lecteur Flash 9 et ActionScript 3 en matière de gestion des objets graphiques.

### Liste interne d’objets enfants

Chaque `DisplayObjectContainer` possède une liste d’objets enfants représentée par un tableau interne. A chaque index se trouve un objet enfant. Visuellement l’objet positionné à l’index 0 est en dessous de l’objet positionné à l’index 1.

La figure 3-5 illustre le concept :



*Figure 3-5. Tableau interne d’accès aux objets enfants.*

Ce `DisplayObjectContainer` contient trois objets enfants. A chaque appel, la méthode `addChild` place l’objet graphique concerné à la fin de la liste d’objets enfants, ce qui se traduit par un empilement de ces derniers. En travaillant avec les différentes méthodes de gestion des objets enfants nous travaillerons de manière transparente avec ce tableau interne.

La méthode `addChild` empile les objets enfants, mais si nous souhaitons gérer l’ordre d’empilement, nous utiliserons la méthode `addChildAt` dont voici la signature :

```
public function addChildAt(child:DisplayObject, index:int):DisplayObject
```

La méthode `addChildAt` accepte un deuxième paramètre permettant de gérer l’index du `DisplayObject` au sein de la liste du `DisplayObjectContainer`.

Prenons un exemple simple, nous créons une instance de `MovieClip` et l’ajoutons à la liste d’affichage :

```
var monClipA:MovieClip = new MovieClip();  
addChild ( monClipA );
```

L’objet graphique `monClipA` est aussitôt placé à l’index 0. Nous souhaitons alors placer un deuxième clip au même index.

La méthode `addChildAt` place le clip `monClipB` à l’index 0 déplaçant `monClipA` à l’index 1 :

```
var monClipA:MovieClip = new MovieClip();  
addChild ( monClipA );  
var monClipB:MovieClip = new MovieClip();  
addChildAt ( monClipB, 0 );
```

Afin de faire place à l’objet graphique ajouté, les objets enfants déjà présents à l’index spécifié ne sont pas remplacés mais déplacés d’un index.

---

Ce comportement évite d’écraser par erreur un objet graphique présent au sein de la liste d’affichage.

Si nous souhaitons reproduire le mécanisme d’écrasement, nous devons supprimer l’objet graphique puis en placer un nouveau à l’index libéré.

---

Si l’index passé à la méthode `addChildAt` est négatif ou ne correspond à aucun objet enfant :

```
var monClipA:MovieClip = new MovieClip();  
addChildAt ( monClipA, 10 );
```

Une erreur à l’exécution de type `RangeError` est levée :

```
| RangeError: Error #2006: L'index indiqué sort des limites.
```

La méthode `addChildAt` ne peut donc être utilisée qu’avec un index allant de 0 à `numChildren-1`. Pour placer un `DisplayObject` devant tous les autres nous pouvons écrire :

```
| addChildAt ( monClipA, numChildren-1 );
```

Aucune profondeur intermédiaire entre deux `DisplayObject` ne peut demeurer inoccupée. Ce comportement est lié la gestion de la profondeur automatique par le lecteur 9 en ActionScript 3.

Nous reviendrons plus tard sur cette notion dans la partie intitulée *Effondrement des profondeurs*.

## A retenir

- Il existe une seule et unique liste d’affichage.
- En son sommet se trouve l’objet `Stage`.
- Tous les objets graphiques s’instancient avec le mot clé `new`
- Aucune profondeur ou nom d’occurrence n’est nécessaire durant la phase d’instanciation.
- Lorsqu’un objet graphique est créé, il n’est pas ajouté automatiquement à la liste d’affichage.
- Pour voir un objet graphique il faut l’ajouter à la liste d’affichage.
- Pour choisir la position du `DisplayObject` lors de l’ajout à la liste d’affichage, nous utilisons la méthode `addChildAt`.

## Accéder aux objets d’affichage

Ouvrez un nouveau document Flash CS3 et créez une simple forme vectorielle à l’aide de l’outil Rectangle. En ciblant la propriété `numChildren` de notre scène principale, nous récupérerons le nombre d’enfants correspondant à la longueur du tableau interne :

```
// affiche : 1  
trace( numChildren );
```

La propriété `numChildren` nous renvoie 1, car le seul objet contenu par notre scène principale est notre forme vectorielle. Celle-ci a été ajoutée automatiquement à la liste d’affichage. Bien que la forme vectorielle n’ait pas de nom d’occurrence nous pouvons y accéder grâce aux méthodes définies par la classe `DisplayObjectContainer`.

Pour accéder à un objet enfant placé à index spécifique nous avons à disposition la méthode `getChildAt` :

```
public function getChildAt(index:int):DisplayObject
```

N’oubliez pas que la méthode `getChildAt` ne fait que pointer dans le tableau interne du `DisplayObjectContainer` selon l’index passé en paramètre :

```
monDisplayObjectContainer.getChildAt ( index );
```



Pour accéder à la forme vectorielle que nous venons de dessiner, nous ciblons l’index 0 à l’aide de la méthode `getChildAt` :

```
// accède à l'objet graphique placé à l'index 0
var forme:Shape = getChildAt ( 0 );
```

En compilant le code précédent, une erreur est levée nous indiquant qu’il est impossible de placer un objet de type `DisplayObject` au sein d’une variable de type `Shape` :

```
1118: Contrainte implicite d'une valeur du type statique
flash.display:DisplayObject vers un type peut-être sans rapport
flash.display:Shape.
```

Nous tentons de stocker la référence renvoyée par la méthode `getChildAt` au sein d’un conteneur de type `Shape`. En relisant la signature de la méthode `getChildAt` nous pouvons lire que le type retourné par celle-ci est `DisplayObject`.

Flash refuse alors la compilation car nous tentons de stocker un objet de type `DisplayObject` dans un conteneur de type `Shape`.

Nous pouvons donc indiquer au compilateur que l’objet sera bien un objet de type `Shape` en utilisant le mot clé `as` :

```
// accède à l'objet graphique placé à l'index 0
var forme:Shape = getChildAt ( 0 ) as Shape;

// affiche : [object Shape]
trace( forme );
```

En réalité, ce code s’avère dangereux car si l’objet placé à l’index 0 est remplacé par un objet graphique non compatible avec le type `Shape`, le résultat du transtypage échoue et renvoie `null`.

Nous devons donc nous assurer que quelque soit l’objet graphique placé à l’index 0, notre variable soit de type compatible. Dans cette situation, il convient donc de **toujours** utiliser le type générique `DisplayObject` pour stocker la référence à l’objet graphique :

```
// accède à l'objet graphique placé à l'index 0
var forme:DisplayObject = getChildAt ( 0 );

// affiche : [object Shape]
trace( forme );
```

De cette manière, nous ne prenons aucun risque, car de par l’héritage, l’objet graphique sera forcément de type `DisplayObject`.

Si nous tentons d’accéder à un index inoccupé :

```
// accède à l'objet graphique placé à l'index 0
var forme:DisplayObject = getChildAt ( 1 );
```

Une erreur de type `RangeError` est levée, car l’index 1 spécifié ne contient aucun objet graphique :

```
RangeError: Error #2006: L'index indiqué sort des limites.
```

Avant ActionScript 3 lorsqu’un objet graphique était créé à une profondeur déjà occupée, l’objet résidant à cette profondeur était remplacé. Pour éviter tout conflit entre graphistes et développeurs les objets graphiques créés par programmation étaient placés à une profondeur positive et ceux créés dans l’environnement auteur à une profondeur négative.

Ce n’est plus le cas en ActionScript 3, tout le monde travaille dans le même niveau de profondeurs. Lorsque nous posons un objet graphique depuis l’environnement auteur de Flash CS3 sur la scène, celui-ci est créé et automatiquement ajouté à la liste d’affichage.

---

En ActionScript 1 et 2 il était impossible d’accéder à une simple forme contenue au sein d’un scénario. Une des forces d’ActionScript 3 est la possibilité de pouvoir accéder à n’importe quel objet de la liste d’affichage.

---

Allons un peu plus loin, et créez sur le scénario principal une seconde forme vectorielle à l’aide de l’outil Ovale, le code suivant nous affiche toujours au total un seul objet enfant :

```
// affiche : 1  
trace( numChildren );
```

Nous aurions pu croire que chaque forme vectorielle créée correspond à un objet graphique différent, il n’en est rien. Toutes les formes vectorielles créées dans l’environnement auteur ne sont en réalité qu’un seul objet `Shape`.

---

Même si les formes vectorielles sont placées sur plusieurs calques, un seul objet `Shape` est créé pour contenir la totalité des tracés.

---

Le code suivant rend donc invisible nos deux formes vectorielles :

```
var mesFormes:DisplayObject = getChildAt ( 0 );  
  
mesFormes.visible = false;
```

Rappelez-vous, ActionScript 3 intègre une gestion de la profondeur automatique. A chaque appel de la méthode `addChild` le `DisplayObject` passé en paramètre est ajouté au `DisplayObjectContainer`. Les objets enfants s’ajoutant les uns derrière les autres au sein du tableau interne.

Chaque objet graphique est placé graphiquement au dessus du précédent. Il n’est donc plus nécessaire de se soucier de la profondeur d’un `MovieClip` au sein d’une boucle `for` :

```
var monClip:MovieClip;

var i:int;

for ( i = 0; i< 10; i++ )
{
    monClip = new MovieClip();

    addChild ( monClip );
}
```

En sortie de boucle, dix clips auront été ajoutés à la liste d’affichage :

```
var monClip:MovieClip;

var i:int;

for ( i = 0; i< 10; i++ )
{
    monClip = new MovieClip();

    addChild ( monClip );
}

// affiche : 10
trace( numChildren );
```

Pour faire référence à chaque clip nous pouvons ajouter le code suivant :

```
var lng:int = numChildren;

for ( i = 0; i< lng; i++ )
{
    /* affiche :
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    [object MovieClip]
    */
    trace( getChildAt ( i ) );
}
```

Nous récupérons le nombre total d’objets enfants avec la propriété `numChildren`, puis nous passons l’indice `i` comme index à la méthode `getChildAt` pour cibler chaque occurrence.

Pour récupérer l’index associé à un `DisplayObject`, nous le passons en référence à la méthode `getChildIndex` :

```
public function getChildIndex(child:DisplayObject):int
```

Dans le code suivant nous accédons aux clips précédemment créés et récupérons l’index de chacun :

```
var lng:int = numChildren;

var objetEnfant:DisplayObject;

for ( i = 0; i< lng; i++ )
{
    objetEnfant = getChildAt( i );

    /*affiche :
    [object MovieClip] à l'index : 0
    [object MovieClip] à l'index : 1
    [object MovieClip] à l'index : 2
    [object MovieClip] à l'index : 3
    [object MovieClip] à l'index : 4
    [object MovieClip] à l'index : 5
    [object MovieClip] à l'index : 6
    [object MovieClip] à l'index : 7
    [object MovieClip] à l'index : 8
    [object MovieClip] à l'index : 9
    */
    trace ( objetEnfant + " à l'index : " + getChildIndex ( objetEnfant ) );
}
```

Sachez qu’il est toujours possible de donner un nom à un objet graphique. En ActionScript 1 et 2 pour donner un nom à un clip, il fallait spécifier le nom de l’occurrence en tant que paramètre lors de l’appel de la méthode `createEmptyMovieClip` ou `attachMovie`.

En ActionScript 3, une fois l’objet graphique instancié nous pouvons passer une chaîne de caractères à la propriété `name` définie par la classe `DisplayObject`.

Le code suivant crée une instance de `MovieClip` et lui affecte un nom d’occurrence :

```
var monClip:MovieClip = new MovieClip();

monClip.name = "monOccurrence";

addChild ( monClip );

// affiche : monOccurrence
```

```
| trace(monClip.name);
```

Pour accéder à notre `MovieClip` nous pouvons utiliser la méthode `getChildByName` permettant de cibler un `DisplayObject` par son nom et non par son index comme le permet la méthode `getChildAt`.

La méthode `getChildByName` définie par la classe `DisplayObjectContainer` accepte comme seul paramètre une chaîne de caractères correspondant au nom d’occurrence du `DisplayObject` auquel nous souhaitons accéder. Voici sa signature :

```
| public function getChildByName(name:String):DisplayObject
```

Le code suivant nous permet de cibler notre `MovieClip` :

```
| var monClip:MovieClip = new MovieClip();
| monClip.name = "monOccurrence";
| addChild ( monClip );
|
| // affiche : [object MovieClip]
| trace( getChildByName ("monOccurrence") );
```

La méthode `getChildByName` traverse récursivement tous les objets de la liste d’objets enfants jusqu’à ce que le `DisplayObject` soit trouvé.

Si l’objet graphique recherché n’est pas présent dans le `DisplayObjectContainer` concerné, la méthode `getChildByName` renvoie `null` :

```
| var monClip:MovieClip = new MovieClip();
| monClip.name = "monOccurrence";
|
| // affiche : null
| trace( getChildByName ("monOccurrence") );
```

Il n’est pas recommandé d’utiliser la méthode `getChildByName`, celle-ci s’avère beaucoup plus lente à l’exécution que la méthode `getChildAt` qui pointe directement dans le tableau interne du `DisplayObjectContainer`.

Un simple test met en évidence la différence significative de performances. Au sein d’une boucle nous accédons à un clip présent au sein de la liste d’affichage à l’aide de la méthode `getChildByName` :

```
| var monClip:MovieClip = new MovieClip();
| monClip.name = "monOccurrence";
|
| addChild ( monClip );
```

```
var depart:Number = getTimer();
for ( var i:int = 0; i< 5000000; i++ )

{

    getChildByName ("monOurrence");

}

var arrivee:Number = getTimer();

// affiche : 854 ms
trace( (arrivee - depart) + " ms" );
```

Notre boucle met environ 854 ms à s’exécuter en utilisant la méthode d’accès `getChildByName`. Procédons au même test en utilisant cette fois la méthode d’accès `getChildAt` :

```
var monClip:MovieClip = new MovieClip();

monClip.name = "monOurrence";

addChild ( monClip );

var depart:Number = getTimer();

for ( var i:int = 0; i< 5000000; i++ )

{

    getChildAt (0);

}

var arrivee:Number = getTimer();

// affiche : 466 ms
trace( (arrivee - depart) + " ms" );
```

Nous obtenons une différence d’environ 400 ms entre les deux boucles. Le bilan de ce test nous pousse à utiliser la méthode `getChildAt` plutôt que la méthode `getChildByName`.

Une question que certains d’entre vous peuvent se poser est la suivante : Si je ne précise pas de nom d’occurrence, quel nom porte mon objet graphique ?

Le lecteur Flash 9 procède de la même manière que les précédents lecteurs en affectant un nom d’occurrence par défaut.

Prenons l’exemple d’un `MovieClip` créé dynamiquement :

```
var monClip:MovieClip = new MovieClip();

// affiche : instance1
trace( monClip.name );
```

Un nom d’occurrence est automatiquement affecté.

Il est important de noter qu’il est impossible de modifier la propriété `name` d’un objet créé depuis l’environnement auteur.

Dans le code suivant, nous tentons de modifier la propriété `name` d’un `MovieClip` créé depuis l’environnement auteur :

```
| monClip.name = "nouveauNom";
```

Ce qui génère l’erreur suivante à la compilation :

```
| Error: Error #2078: Impossible de modifier la propriété de nom d'un objet placé  
sur le scénario.
```

En pratique, cela ne pose pas de réel problème car nous utiliserons très rarement la propriété `name` qui est fortement liée à la méthode `getChildByName`.

## Suppression d’objets d’affichage

Pour supprimer un `DisplayObject` de l’affichage nous appelons la méthode `removeChild` sur son conteneur, un objet `DisplayObjectContainer`. La méthode `removeChild` prend comme paramètre le `DisplayObject` à supprimer de la liste d’affichage et renvoie sa référence :

```
| public function removeChild(child:DisplayObject):DisplayObject
```

Il est essentiel de retenir que la suppression d’un objet graphique est toujours réalisée par l’objet parent. La méthode `removeChild` est donc toujours appelée sur l’objet conteneur :

```
| monConteneur.removeChild ( enfant );
```

De ce fait, un objet graphique n’est pas en mesure de se supprimer lui-même comme c’était le cas avec la méthode `removeMovieClip` existante en ActionScript 1 et 2.

Il est en revanche capable de demander à son parent de le supprimer :

```
| parent.removeChild ( this );
```

Il est important de retenir que l’appel de la méthode `removeChild` procède à une simple suppression du `DisplayObject` au sein de la liste d’affichage mais ne le détruit pas.

---

Un ancien réflexe lié à ActionScript 1 et 2 pourrait vous laisser penser que la méthode `removeChild` est l’équivalent de la méthode `removeMovieClip`, ce n’est pas le cas.

---

Pour nous rendre compte de ce comportement, ouvrez un nouveau document Flash CS3 et créez un symbole de type clip. Posez une

occurrence de ce dernier sur le scénario principal et nommez la `monRond`.

Sur un calque AS nous ajoutons un appel à la méthode `removeChild` pour supprimer le clip de la liste d’affichage.

```
removeChild ( monRond );
```

Nous pourrions penser que notre clip `monRond` a aussi été supprimé de la mémoire. Mais si nous ajoutons la ligne suivante :

```
removeChild ( monRond );  
  
// affiche : [object MovieClip]  
trace( monRond );
```

Nous voyons que le clip `monRond` existe toujours et n’a pas été supprimé de la mémoire. En réalité nous avons supprimé de l’affichage notre clip `monRond`. Ce dernier n’est plus rendu mais continue d’exister en mémoire.

---

Si le développeur ne prend pas en compte ce mécanisme, les performances d’une application peuvent être mises en péril.

---

Prenons le cas classique suivant : un événement `Event.ENTER_FRAME` est écouté auprès d’une instance de `MovieClip` :

```
var monClip:MovieClip = new MovieClip();  
  
monClip.addEventListener( Event.ENTER_FRAME, ecouteur );  
  
addChild ( monClip );  
  
function ecouteur ( pEvt:Event ):void  
{  
  
    trace("exécution");  
  
}
```

En supprimant l’instance de `MovieClip` de la liste d’affichage à l’aide de la méthode `removeChild`, nous remarquons que l’événement est toujours diffusé :

```
var monClip:MovieClip = new MovieClip();  
  
monClip.addEventListener( Event.ENTER_FRAME, ecouteur );  
  
addChild ( monClip );  
  
function ecouteur ( pEvt:Event ):void  
{
```



```
        trace("exécution");
    }
    removeChild ( monClip );
```

Pour libérer de la mémoire un `DisplayObject` supprimé de la liste d’affichage nous devons passer ses références à `null` et attendre le passage du ramasse-miettes. Rappelez-vous que ce dernier supprime les objets n’ayant plus aucune référence au sein de l’application.

Le code suivant supprime notre clip de la liste d’affichage et passe sa référence à `null`. Ce dernier n’est donc plus référencé au sein de l’application le rendant donc éligible pour le ramasse miettes :

```
var monClip:MovieClip = new MovieClip();
monClip.addEventListener( Event.ENTER_FRAME, ecouteur );
addChild ( monClip );

function ecouteur ( pEvt:Event ):void
{
    trace("exécution");
}

removeChild ( monClip );

monClip = null;
// affiche : null
trace( monClip );
```

Bien que nous ayons supprimé toutes les références vers notre `MovieClip`, celui-ci n’est pas supprimé de la mémoire immédiatement.

Rappelez-vous que le passage du ramasse-miettes est différé !

Nous ne pouvons pas savoir quand ce dernier interviendra. Le lecteur gère cela de manière autonome selon les ressources système.

---

Il est donc impératif de prévoir un mécanisme de désactivation des objets graphiques, afin que ces derniers consomment un minimum de ressources lorsqu’ils ne sont plus affichés et attendent d’être supprimés par le ramasse-miettes.

---

Durant nos développements, nous pouvons néanmoins déclencher manuellement le passage du ramasse-miettes au sein du lecteur de débogage à l’aide de la méthode `gc` de la classe `System`.

Dans le code suivant, nous déclenchons le ramasse-miettes lors du clic souris sur la scène :

```
var monClip:MovieClip = new MovieClip();

monClip.addEventListener( Event.ENTER_FRAME, ecouteur );

addChild ( monClip );

function ecouteur ( pEvt:Event ):void
{
    trace("exécution");
}

removeChild ( monClip );

monClip = null;

stage.addEventListener ( MouseEvent.CLICK, clicSouris );

function clicSouris ( pEvt:MouseEvent ):void
{
    // déclenchement du ramasse-miettes
    System.gc();
}
```

En cliquant sur la scène, l’événement `Event.ENTER_FRAME` cesse d’être diffusé car le passage du ramasse-miettes a entraîné une suppression définitive du `MovieClip`.

Nous reviendrons sur ce point dans une prochaine partie intitulée *Désactivation des objets graphiques*.

Imaginons le cas suivant : nous devons supprimer un ensemble de clips que nous venons d’ajouter à la liste d’affichage.

Ouvrez un nouveau document Flash CS3 et placez plusieurs occurrences de symbole `MovieClip` sur la scène principale. Il n’est pas nécessaire de leur donner des noms d’occurrences.

En ActionScript 1 et 2, nous avons plusieurs possibilités. Une première technique consistait à stocker dans un tableau les références aux clips ajoutés, puis le parcourir pour appeler la méthode `removeMovieClip` sur chaque référence.

Autre possibilité, parcourir le clip conteneur à l’aide d’une boucle `for in` pour récupérer chaque propriété faisant référence aux clips pour pouvoir les cibler puis les supprimer.

Lorsque vous souhaitez supprimer un objet graphique positionné à un index spécifique nous pouvons utiliser la méthode `removeChildAt` dont voici la signature :

```
| public function removeChildAt(index:int):DisplayObject
```

En découvrant cette méthode nous pourrions être tentés d’écrire le code suivant :

```
| var lng:int = numChildren;  
|  
| for ( var i:int = 0; i< lng; i++ )  
|  
| {  
|  
|     removeChildAt ( i );  
|  
| }
```

A l’exécution le code précédent génère l’erreur suivante :

```
| RangeError: Error #2006: L'index indiqué sort des limites.
```

Une erreur de type `RangeError` est levée car l’index que nous avons passé à la méthode `getChildIndex` est considéré comme hors-limite. Cette erreur est levée lorsque l’index passé ne correspond à aucun objet enfant contenu par le `DisplayObjectContainer`.

Notre code ne peut pas fonctionner car nous n’avons pas pris en compte un comportement très important en ActionScript 3 appelé *effondrement des profondeurs*.

|

## A retenir

|

- Chaque `DisplayObjectContainer` contient un tableau interne contenant une référence à chaque objet enfant.
- Toutes les méthodes de manipulation des objets d’affichage travaillent avec le tableau interne d’objets enfants de manière transparente.
- La profondeur est gérée automatiquement.
- La méthode la plus rapide pour accéder à un objet enfant est `getChildAt`
- La méthode `addChild` ne nécessite aucune profondeur et empile chaque objet enfant.
- La méthode `removeChild` supprime un `DisplayObject` de la liste d’affichage mais ne le détruit pas.
- Pour supprimer un `DisplayObject` de la mémoire, nous devons passer ses références à `null`.
- Le ramasse miettes interviendra quand il le souhaite et supprimera les objets non référencés de la mémoire.

## Effondrement des profondeurs

En ActionScript 1 et 2, lorsqu’un objet graphique était supprimé, les objets placés au dessus de ce dernier conservaient leur profondeur. Ce comportement paraissait tout à fait logique mais soulevait un problème d’optimisation car des profondeurs demeuraient inoccupées entre les objets graphiques. Ainsi les profondeurs 6, 8 et 30 pouvaient être occupées alors que les profondeurs intermédiaires restaient inoccupées.

---

En ActionScript 3 aucune profondeur intermédiaire ne peut rester vide au sein de la liste d’affichage.

---

Pour comprendre le concept d’effondrement des objets d’affichage prenons un exemple de la vie courante, une pile d’assiettes. Si nous retirons une assiette située en bas de la pile, les assiettes situées au dessus de celle-ci descendront d’un niveau. Au sein de la liste d’affichage, les objets d’affichage fonctionnent de la même manière.

Lorsque nous supprimons un objet d’affichage, les objets situés au dessus descendent alors d’un niveau. On dit que les objets d’affichage s’effondrent. De cette manière aucune profondeur entre deux objets d’affichage ne peut demeurer inoccupée.

En prenant en considération cette nouvelle notion, relisons notre code censé supprimer la totalité des objets enfants contenus dans un `DisplayObjectContainer` :

```
var lng:int = numChildren;
```

```
for ( var i:int = 0; i< lng; i++ )  
{  
    removeChildAt ( i );  
}
```

A chaque itération la méthode `removeChildAt` supprime un objet enfant, faisant descendre d’un index tous les objets enfants supérieurs. L’objet en début de pile étant supprimé, le suivant devient alors le premier de la liste d’objets enfants, le même processus se répète alors pour chaque itération.

En milieu de boucle, notre index pointe donc vers un index inoccupé levant une erreur de type `RangeError`.

En commençant par le haut de la pile nous n’entraînons aucun effondrement de pile. Nous pouvons supprimer chaque objet de la liste :

```
var lng:int = numChildren-1;  
for ( var i:int = lng; i>= 0; i-- )  
{  
    removeChildAt ( i );  
}  
  
// affiche : 0  
trace( numChildren );
```

La variable `lng` stocke l’index du dernier objet enfant, puis nous supprimons chaque objet enfant en commençant par le haut de la pile puis en descendant pour chaque itération.

Cette technique fonctionne sans problème, mais une approche plus concise et plus élégante existe.

Ne considérons pas l’effondrement automatique comme un inconvénient mais plutôt comme un avantage, nous pouvons écrire :

```
while ( numChildren > 0 )  
{  
    removeChildAt ( 0 );  
}  
  
// affiche : 0  
trace( numChildren );
```

Nous supprimons à chaque itération l’objet graphique positionné à l’index 0. Du fait de l’effondrement, chaque objet enfant passera forcément par cet index.

### A retenir

- La suppression d’un `DisplayObject` au sein de la liste d’affichage provoque un effondrement des profondeurs.
- L’effondrement des profondeurs permet de ne pas laisser de profondeurs intermédiaires inoccupées entre les objets graphiques.

### Gestion de l’empilement des objets d’affichage

Pour modifier la superposition d’objets enfants au sein d’un `DisplayObjectContainer`, nous disposons de la méthode `setChildIndex` dont voici la signature :

```
public function setChildIndex(child:DisplayObject, index:int):void
```

La méthode `setChildIndex` est appelée sur le conteneur des objets enfants à manipuler. Si l’index passé est négatif ou ne correspond à aucun index occupé, son appel lève une exception de type `RangeError`.

Pour bien comprendre comment fonctionne cette méthode, passons à un peu de pratique.

Ouvrez un nouveau document Flash CS3 et créez deux clips de formes différentes. Superposez-les comme l’illustre la figure 3.6



*Figure 3-6. Clips superposés*

Nous donnerons comme noms d’occurrence `monRond` et `monRectangle`. Nous souhaitons faire passer le clip `monRond` devant le clip `monRectangle`. Nous pouvons en déduire facilement que ce dernier est positionné l’index 1, et le premier à l’index 0.

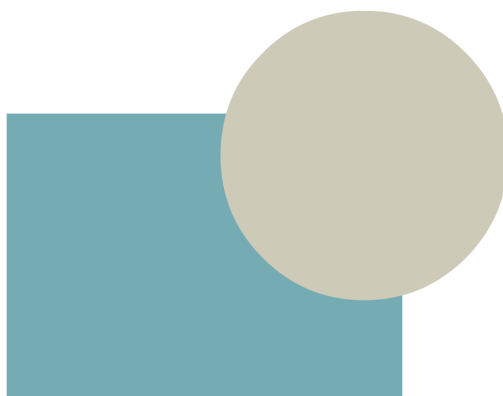
Pour placer le rond à l’index 1 nous le passons à la méthode `setChildIndex` tout en précisant l’index de destination.

Sur un calque AS, nous définissons le code suivant :

```
| setChildIndex( monRond, 1 );
```

Le rond est supprimé temporairement de la liste faisant chuter notre rectangle à l’index 0 pour laisser place au rond à l’index 1.

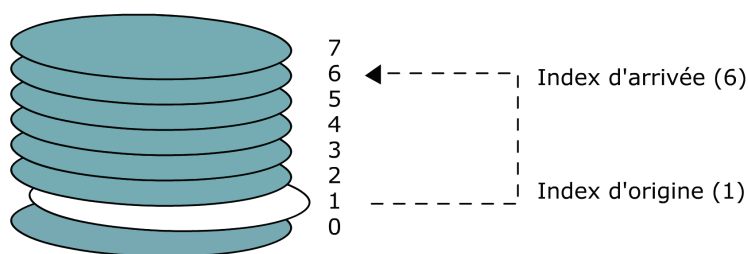
Ce dernier est donc placé en premier plan comme l’illustre la figure 3.7



*Figure 3-7. Changement de profondeurs.*

Le changement d’index d’un `DisplayObject` est divisé en plusieurs phases. Prenons l’exemple de huit objets graphiques. Nous souhaitons passer le `DisplayObject` de l’index 1 à l’index 6.

La figure 3-8 illustre l’idée :



*Figure 3-8. Changement d’index pour un `DisplayObject`.*

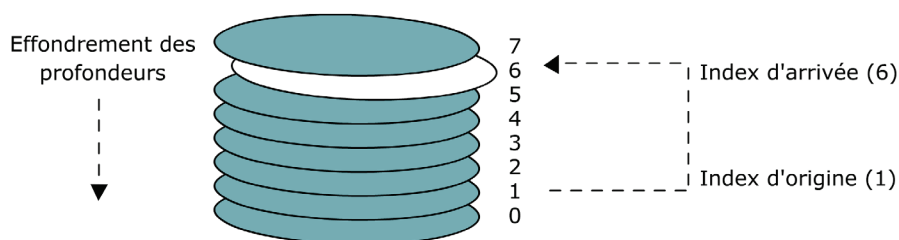
Si nous ne connaissons pas le nom des objets graphiques, nous pouvons écrire :

```
| setChildIndex ( getChildAt ( 1 ), 6 );
```

Nous récupérons le `DisplayObject` à l’index 1 pour le passer à l’index 6. Lorsque ce dernier est retiré de son index d’origine, les

`DisplayObject` supérieurs vont alors descendre d’un index. Exactement comme si nous retirions une assiette d’une pile d’assiettes. Il s’agit du comportement que nous avons évoqué précédemment appelé effondrement des profondeurs.

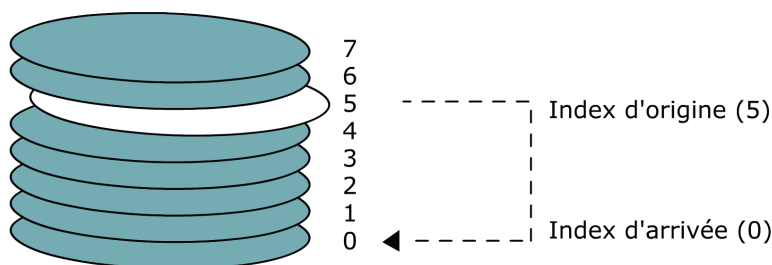
La figure 3-9 illustre le résultat du changement d’index du `DisplayObject` :



*Figure 3-9. Effondrement des profondeurs.*

Prenons une autre situation. Cette fois nous souhaitons déplacer un `DisplayObject` de l’index 5 à l’index 0.

La figure 3-10 illustre l’idée :



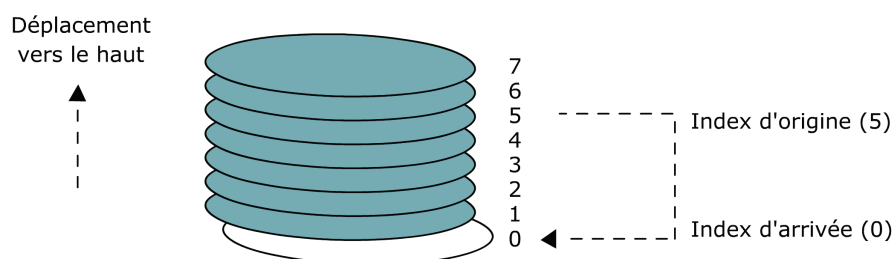
*Figure 3-10. Déplacement vers le bas de la pile.*

Pour procéder à ce déplacement au sein de la liste d’objets enfants nous pouvons écrire :

```
setChildIndex ( monDisplayObject, 0 );
```

Lorsqu’un `DisplayObject` est déplacé à un index inférieur à son index d’origine, les objets graphiques intermédiaires ne s’effondrent pas mais remontent d’un index. Le code précédent provoque donc l’organisation présentée par la figure 3-11 :





*Figure 3-11. Objets graphiques poussés d'un index vers le haut de la pile.*

Lors de l'utilisation de la méthode `setChildIndex` les objets graphiques intermédiaires compris entre l'index d'origine et d'arrivée peuvent être poussés ou descendus d'un niveau.

## Echange de profondeurs

Dans certaines situations nous pouvons avoir besoin d'intervertir l'ordre d'empilement de deux objets graphiques. Pour cela nous utiliserons les méthodes `swapChildren` ou `swapChildrenAt`. Nous allons nous attarder sur la première dont voici la signature :

```
| public function swapChildren(child1:DisplayObject, child2:DisplayObject):void
```

La méthode `swapChildren` accepte deux paramètres de type `DisplayObject`. Contrairement à la méthode `setChildIndex`, les méthodes `swapChildren` et `swapChildrenAt` ne modifient pas l'index des autres objets graphiques lors de l'échange de profondeur.

Nous appelons la méthode `swapChildren` sur notre scénario principal qui contient nos deux objets graphiques. Le code suivant procède à l'échange des profondeurs :

```
| swapChildren ( monRond, monRectangle );
```

Si nous testons notre animation, nous obtenons l'affichage illustré par la figure 3-12.



*Figure 3-12. Echange des profondeurs.*

Un appel successif de la méthode `swapChildren` intervertit à chaque appel les deux `DisplayObject`. Si nous rajoutons un appel à la méthode `swapChildren` nous obtenons l’affichage de départ.

Sur notre calque, nous rajoutons un deuxième appel à la méthode `swapChildren` :

```
swapChildren ( monRond, monRectangle );  
  
swapChildren ( monRond, monRectangle );
```

Nous obtenons l’organisation de départ comme l’illustre la figure 3-13 :



*Figure 3-13. Organisation de départ.*

Lorsque nous n’avons pas de référence aux `DisplayObject` à intervertir nous pouvons utiliser la méthode `swapChildrenAt`. Cette dernière accepte deux index en paramètres :

```
public function swapChildrenAt(index1:int, index2:int):void
```

Si l’index est négatif ou n’existe pas dans la liste d’enfants, l’appel de la méthode lèvera une exception de type `RangeError`.

Nous pouvons ainsi arriver au même résultat que notre exemple précédent sans spécifier de nom d’occurrence mais directement l’index des `DisplayObject` :

```
swapChildrenAt ( 1, 0 );
```

Si nous avons seulement une référence et un index nous pourrions obtenir le même résultat à l’aide des différentes méthodes que nous avons abordées :

```
swapChildrenAt ( getChildIndex (monRectangle), 0 );
```

Ou encore, même si l’intérêt est moins justifié :

```
swapChildren ( getChildAt (1), getChildAt (0) );
```

## A retenir

- Pour changer l’ordre d’empilement des `DisplayObject` au sein de la liste d’objets enfants, nous utilisons la méthode `setChildIndex`.
- La méthode `setChildIndex` pousse d’un index vers le haut ou vers le bas les autres objets graphiques de la liste d’enfants.
- Pour intervertir l’ordre d’empilement de deux `DisplayObject`, les méthodes `swapChildren` et `swapChildrenAt` seront utilisées.
- Les méthodes `swapChildren` et `swapChildrenAt` ne modifient pas l’ordre d’empilements des autres objets graphiques de la liste d’objets enfants.

## Désactivation des objets graphiques

Lors du développement d’applications ActionScript 3, il est **essentiel** de prendre en considération la désactivation des objets graphiques.

Comme nous l’avons vu précédemment, lorsqu’un objet graphique est supprimé de la liste d’affichage, ce dernier continue de « vivre ».

Afin de désactiver un objet graphique supprimé de la liste d’affichage, nous utilisons les deux événements suivants :

- `Event.ADDED_TO_STAGE` : diffusé lorsqu’un objet graphique est affiché.
- `Event.REMOVED_FROM_STAGE` : diffusé lorsqu’un objet graphique est supprimé de l’affichage.

Ces deux événements extrêmement utiles sont diffusés automatiquement lorsque le lecteur affiche ou supprime de l’affichage un objet graphique.

Dans le code suivant, la désactivation de l’objet graphique n’est pas gérée, lorsque le `MovieClip` est supprimé de l’affichage l’événement `Event.ENTER_FRAME` est toujours diffusé :

```
var monClip:MovieClip = new MovieClip();
monClip.addEventListener( Event.ENTER_FRAME, ecouteur );
addChild ( monClip );

function ecouteur ( pEvt:Event ):void
{
    trace("exécution");
}
```

```
}  
  
removeChild ( monClip );
```

En prenant en considération la désactivation de l’objet graphique, nous écoutons l’événement `Event.REMOVED_FROM_STAGE` afin de supprimer l’écoute de l’événement `Event.ENTER_FRAME` lorsque l’objet est supprimé de l’affichage :

```
var monClip:MovieClip = new MovieClip();  
  
monClip.addEventListener( Event.ENTER_FRAME, ecouteur );  
  
// écoute de l'événement Event.REMOVED_FROM_STAGE  
monClip.addEventListener( Event.REMOVED_FROM_STAGE, desactivation );  
  
addChild ( monClip );  
  
function ecouteur ( pEvt:Event ):void  
{  
    trace("exécution");  
}  
  
function desactivation ( pEvt:Event ):void  
{  
    // suppression de l'écoute de l'événement Event.ENTER_FRAME  
    pEvt.target.removeEventListener ( Event.ENTER_FRAME, ecouteur );  
}  
  
stage.addEventListener ( MouseEvent.CLICK, supprimeAffichage );  
  
function supprimeAffichage ( pEvt:MouseEvent ):void  
{  
    // lors de la suppression de l'objet graphique  
    // l'événement Event.REMOVED_FROM_STAGE est automatiquement  
    // diffusé par l'objet  
    removeChild ( monClip );  
}
```

De cette manière, lorsque l’objet graphique n’est pas affiché, ce dernier ne consomme quasiment aucune ressource car nous avons pris le soin de supprimer l’écoute de tous les événements consommateurs des ressources.

A l’inverse, nous pouvons écouter l’événement `Event.ADDED_TO_STAGE` pour ainsi gérer l’activation et la désactivation de l’objet graphique :

```
var monClip:MovieClip = new MovieClip();
```

---

```
// écoute de l'événement Event.REMOVED_FROM_STAGE
monClip.addEventListener( Event.REMOVED_FROM_STAGE, desactivation );

// écoute de l'événement Event.ADDED_TO_STAGE
monClip.addEventListener( Event.ADDED_TO_STAGE, activation );

addChild ( monClip );

function ecouteur ( pEvt:Event ):void
{
    trace("exécution");
}

function activation ( pEvt:Event ):void
{
    // écoute de l'événement Event.ENTER_FRAME
    pEvt.target.addEventListener ( Event.ENTER_FRAME, ecouteur );
}

function desactivation ( pEvt:Event ):void
{
    // suppression de l'écoute de l'événement Event.ENTER_FRAME
    pEvt.target.removeEventListener ( Event.ENTER_FRAME, ecouteur );
}

stage.addEventListener ( MouseEvent.CLICK, supprimeAffichage );

function supprimeAffichage ( pEvt:MouseEvent ):void
{
    // lors de la suppression de l'objet graphique
    // l'événement Event.REMOVED_FROM_STAGE est automatiquement
    // diffusé par l'objet
    if ( contains ( monClip ) ) removeChild ( monClip );

    // lors de l'affichage de l'objet graphique
    // l'événement Event.ADDED_TO_STAGE est automatiquement
    // diffusé par l'objet
    else addChild ( monClip );
}
```

Au sein de la fonction écouteur `supprimeAffichage`, nous testons si le scénario contient le clip `monClip` à l’aide de la méthode `contains`. Si ce n’est pas le cas nous l’affichons, dans le cas inverse nous le supprimons de l’affichage.

Nous reviendrons sur le concept d’activation et désactivation des objets graphiques tout au long de l’ouvrage.

Afin de totalement désactiver notre `MovieClip` nous devons maintenant supprimer les références pointant vers lui.

Pour cela, nous modifions la fonction `supprimeAffichage` :

```
function supprimeAffichage ( pEvt:MouseEvent ):void
{
    // lors de la suppression de l'objet graphique
    // l'événement Event.REMOVED_FROM_STAGE est automatiquement
    // diffusé par l'objet, l'objet est désactivé
    removeChild ( monClip );

    // suppression de la référence pointant vers le MovieClip
    monClip = null;
}
```

Certains d’entre vous peuvent se demander l’intérêt de ces deux événements par rapport à une simple fonction personnalisée qui se chargerait de désactiver l’objet graphique.

Au sein d’une application, une multitude de fonctions ou méthodes peuvent entraîner la suppression d’un objet graphique. Grâce à ces deux événements, nous savons que, quelque soit la manière dont l’objet est supprimé ou affiché, nous le saurons et nous pourrons gérer son activation et sa désactivation.

Nous allons nous attarder au cours de cette nouvelle partie, au comportement de la tête de lecture afin de saisir tout l’intérêt des deux événements que nous venons d’aborder.

## Fonctionnement de la tête de lecture

Dû au nouveau mécanisme de liste d’affichage du lecteur Flash 9. Le fonctionnement interne de la tête de lecture a lui aussi été modifié.

Ainsi lorsque la tête de lecture rencontre une image ayant des objets graphiques celle-ci ajoute les objets graphiques à l’aide de la méthode `addChild` :

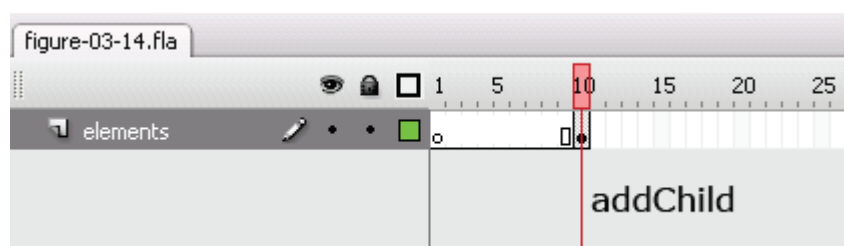


Figure 3-14. Ajout d’objets graphiques.

A l’inverse lorsque la tête de lecture rencontre une image clé vide, le lecteur supprime les objets graphiques à l’aide de la méthode `removeChild` :

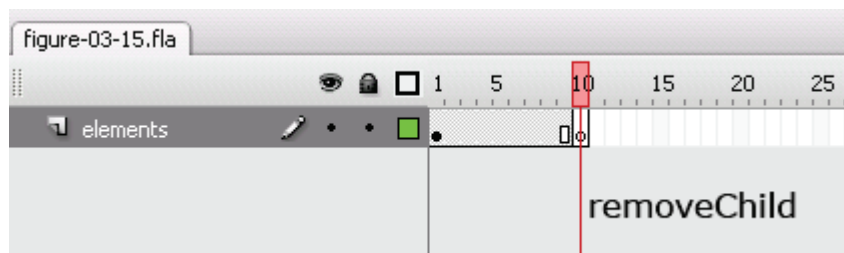


Figure 3-15. Suppression d’objets graphiques.

De par la suppression ou ajout des objets graphiques, ces derniers diffusent automatiquement les événements `Event.ADDED_TO_STAGE` et `Event.REMOVED_FROM_STAGE`.

Nous reviendrons sur ce mécanisme d’activation et désactivation des objets graphiques au cours du chapitre 9 intitulé *Etendre les classes natives*.

## A retenir

- Il est fortement recommandé de gérer l’activation et la désactivation des objets graphiques.
- Pour cela, nous utilisons les événements `Event.ADDED_TO_STAGE` et `Event.REMOVED_FROM_STAGE`.
- Si de nouveaux objets sont placés sur une image clé, la tête de lecture les ajoute à l’aide de la méthode `addChild`.
- Si la tête de lecture rencontre une image clé vide, le lecteur supprime automatiquement les objets graphiques à l’aide de la méthode `removeChild`.

## Subtilités de la propriété `stage`

Nous allons revenir dans un premier temps sur la propriété `stage` définie par la classe `DisplayObject`. Nous avons appris précédemment que l’objet `Stage` n’est plus accessible de manière globale comme c’était le cas en ActionScript 1 et 2.

Pour accéder à l’objet `Stage` nous ciblons la propriété `stage` sur n’importe quel `DisplayObject` :

```
| monDisplayObject.stage
```

La subtilité réside dans le fait que ce `DisplayObject` doit obligatoirement être ajouté à la liste d’affichage afin de pouvoir retourner une référence à l’objet `Stage`.

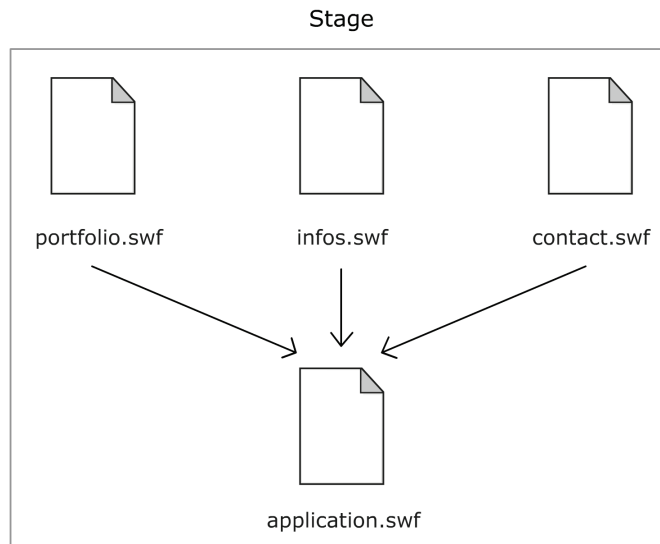
Ainsi, si nous ciblons la propriété `stage` sur un `DisplayObject` non présent dans la liste d’affichage, celle-ci nous renvoie `null` :

```
var monClip:MovieClip = new MovieClip();  
  
// affiche : null  
trace( monClip.stage );
```

Une fois notre clip ajouté à la liste d’affichage, la propriété `stage` contient une référence à l’objet `Stage` :

```
var monClip:MovieClip = new MovieClip();  
  
addChild ( monClip );  
  
// affiche : [object Stage]  
trace( monClip.stage );
```

Grâce à l’événement `Event.ADDED_TO_STAGE`, nous pouvons accéder de manière sécurisée à l’objet `Stage`, car la diffusion de cet événement nous garantit que l’objet graphique est présent au sein de la liste d’affichage.



*Figure 3-16. Objet `Stage`.*

Contrairement aux scénarios qui peuvent être multiples dans le cas de multiples SWF, l’objet `Stage` est unique.



Afin de tester si un objet graphique est affiché actuellement, il suffit donc de tester si sa propriété renvoie `null` ou non.

## A retenir

- La propriété `stage` d’un `DisplayObject` renvoie `null`, tant que ce dernier n’est pas placé au sein de la liste d’affichage.

## Subtilités de la propriété `root`

Les développeurs ActionScript 1 et 2 se souviennent sûrement de la propriété `_root`. En ActionScript 3 son fonctionnement a été entièrement revu, rendant son utilisation totalement sécurisée.

Souvenez vous, la propriété `root` était accessible de manière globale et permettait de référencer rapidement le scénario principal d’une animation. Son utilisation était fortement déconseillée pour des raisons de portabilité de l’application développée.

Prenons un cas typique : nous développons une animation en ActionScript 1 ou 2 qui était plus tard chargée dans un autre SWF. Si nous ciblions notre scénario principal à l’aide de `_root`, une fois l’animation chargée par le SWF, `_root` pointait vers le scénario du SWF chargeur et non plus vers le scénario de notre SWF chargé.

C’est pour cette raison qu’il était conseillé de toujours travailler par ciblage relatif à l’aide de la propriété `_parent` et non par ciblage absolu avec la propriété `_root`.

Comme pour la propriété `stage`, la propriété `root` renvoie `null` tant que le `DisplayObject` n’a pas été ajouté à la liste d’affichage.

```
var monClip:MovieClip = new MovieClip();  
// affiche : null  
trace( monClip.root );
```

Pour que la propriété `root` pointe vers le scénario auquel le `DisplayObject` est rattaché il faut impérativement l’ajouter à la liste d’affichage.

Si l’objet graphique est un enfant du scénario principal, c’est-à-dire contenu par l’objet `MainTimeline`, alors sa propriété `root` pointe vers ce dernier.

Le code suivant est défini sur le scénario principal :

```
var monClip:MovieClip = new MovieClip();  
// le clip monClip est ajouté au scénario principal  
addChild ( monClip );
```

```
// affiche : [object MainTimeline]
trace( monClip.root );
```

Si l’objet graphique n’est pas un enfant du scénario principal, mais un enfant de l’objet `Stage`, sa propriété `root` renvoie une référence vers ce dernier :

```
var monClip:MovieClip = new MovieClip();

// le clip monClip est ajouté à l'objet Stage
stage.addChild ( monClip );

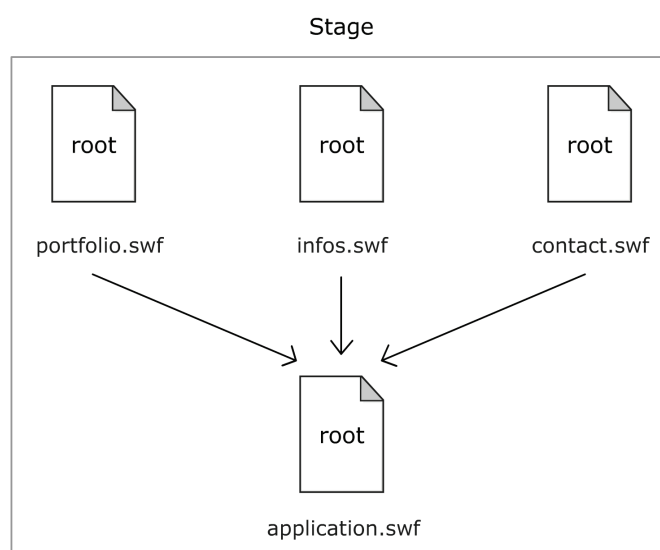
// affiche : [object Stage]
trace( monClip.root );
```

Ainsi, lorsque l’objet graphique n’est pas un enfant du scénario principal, les propriétés `root` et `stage` sont équivalentes car pointent toutes deux vers l’objet `Stage`.

L’usage de la propriété `root` en ActionScript 3 est tout à fait justifié et ne pose aucun problème de ciblage comme c’était le cas en ActionScript 1 et 2.

En ActionScript 3, la propriété `root` référence le scénario principal du SWF en cours. Même dans le cas de chargement externe, lorsqu’un SWF est chargé au sein d’un autre, nous pouvons cibler la propriété `root` sans aucun problème, nous ferons *toujours* référence au scénario du SWF en cours.

La figure 3-17 illustre l’idée :



*Figure 3-17. Scénarios.*

Nous reviendrons sur ce cas plus tard au cours du chapitre 13 intitulé *Chargement de contenu externe*.

## A retenir

- La propriété `root` d’un `DisplayObject` renvoie `null`, tant que ce dernier n’est pas placé au sein de la liste d’affichage.
- Lorsque l’objet graphique n’est pas un enfant du scénario principal, les propriétés `root` et `stage` renvoient toutes deux une référence à l’objet `Stage`.
- La propriété `root` référence le scénario du SWF en cours.
- En ActionScript 3 l’usage de la propriété `root` ne souffre pas des faiblesses existantes dans les précédentes versions d’ActionScript.

## Les `_level`

En ActionScript 3 la notion de `_level` n’est plus disponible, souvenez vous, en ActionScript 1 et 2 nous pouvions écrire :

```
| _level0.createTextField ( "monChampTexte", 0, 0, 0, 150, 25 );
```

Ce code crée un champ texte au sein du `_level0`. En ActionScript 3 si nous tentons d’accéder à l’objet `_level`, nous obtenons une erreur à la compilation :

```
| var monChampTexte:TextField = new TextField();  
| monChampTexte.text = "Bonjour !";  
| _level0.addChild ( monChampTexte );
```

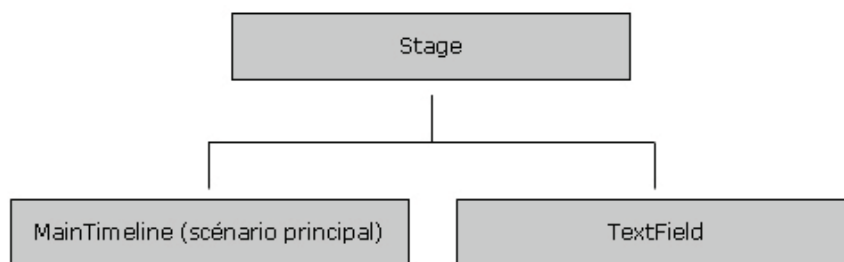
Affiche dans la fenêtre de sortie :

```
| 1120: Accès à la propriété non définie _level0.
```

Il n’existe pas en ActionScript 3 d’objet similaire aux `_level` que nous connaissions. En revanche le même résultat est facilement réalisable en ActionScript 3. En ajoutant nos objets graphiques à l’objet `Stage` plutôt qu’à notre scénario, nous obtenons le même résultat :

```
| var monChampTexte:TextField = new TextField();  
| monChampTexte.text = "Bonjour !";  
| stage.addChild ( monChampTexte );
```

Le code précédent définit la liste d’affichage illustrée par la figure suivante :



*Figure 3-17. Simulation de `_level`.*

Notre champ texte est ici un enfant direct de l’objet `Stage`, nous obtenons donc le même concept d’empilement de scénario établi par les `_level`. Dans cette situation, l’objet `Stage` contient alors deux enfants directs :

```
var monChampTexte:TextField = new TextField();  
  
monChampTexte.text = "Bonjour !";  
  
stage.addChild ( monChampTexte );  
  
// affiche : 2  
trace ( stage.numChildren );
```

Même si cette technique nous permet de simuler la notion de `_level` en ActionScript 3, son utilisation n’est pas recommandée sauf cas spécifique, comme le cas d’une fenêtre d’alerte qui devrait être affichée au dessus de tous les éléments de notre application.

## A retenir

- En ActionScript 3, la notion de `_level` n’existe plus.

# 5

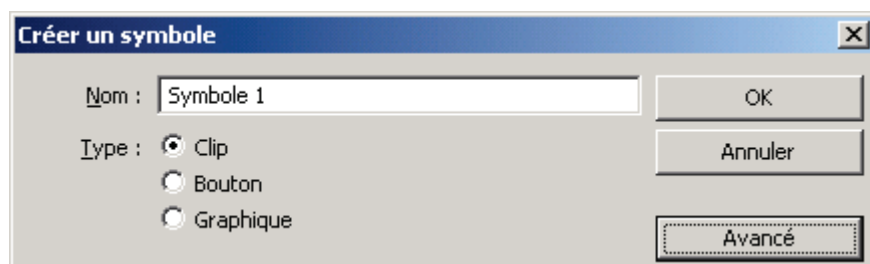
## Symboles prédéfinis

<b>LES TYPES DE SYMBOLE.....</b>	<b>1</b>
LE SYMBOLE CLIP.....	2
LA PROPRIÉTÉ NAME.....	4
<b>INSTANCIATION DE SYMBOLES PAR PROGRAMMATION.....</b>	<b>6</b>
INSTANCIATION DE SYMBOLES PRÉDÉFINIS.....	8
EXTRAIRE UNE CLASSE DYNAMIQUEMENT.....	13
LE SYMBOLE BOUTON.....	14
LE SYMBOLE BOUTON.....	21
LE SYMBOLE GRAPHIQUE.....	22
LES IMAGES BITMAP.....	24
<b>AFFICHER UNE IMAGE BITMAP.....</b>	<b>26</b>
<b>LE SYMBOLE SPRITE.....</b>	<b>28</b>
<b>DEFINITION DU CODE DANS UN SYMBOLE.....</b>	<b>30</b>

### Les types de symbole

Durant nos développements ActionScript nous avons très souvent besoin d'utiliser des symboles prédéfinis. Créés depuis l'environnement auteur de Flash, un logo, une animation, ou encore une image pourront être stockés au sein de la bibliothèque pour une utilisation future durant l'exécution de l'application.

Pour convertir un objet graphique en symbole, nous sélectionnons ce dernier et appuyons sur F8. Le panneau convertir en symbole s'ouvre, comme l'illustre la figure 5.1 :



*Figure 5-1. Panneau Convertir en symbole.*

Trois types de symboles sont proposés :

- Clip
- Bouton
- Graphique

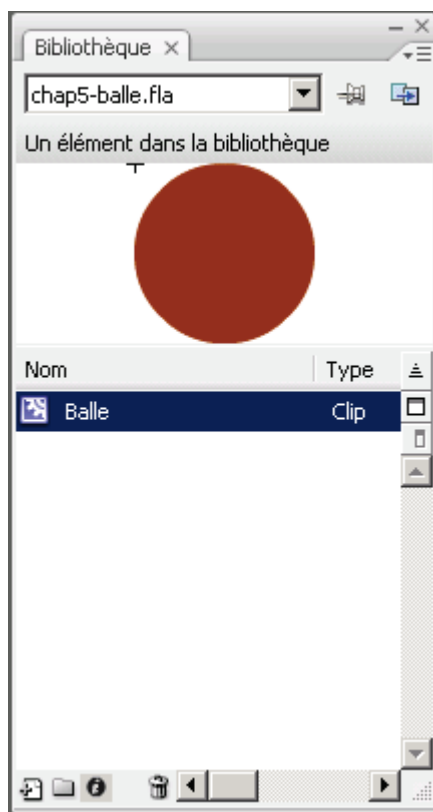
Le type bitmap existe aussi pour les symboles, mais n'apparaît pas ici car il est impossible de créer un bitmap depuis le panneau *Convertir en symbole*. Seules les images importées dans la bibliothèque sont intégrées en tant que type bitmap.

Le symbole clip s'avère être le plus courant dans les développements, nous allons commencer par ce dernier en découvrant les nouveautés apportées par ActionScript 3.

### **Le symbole clip**

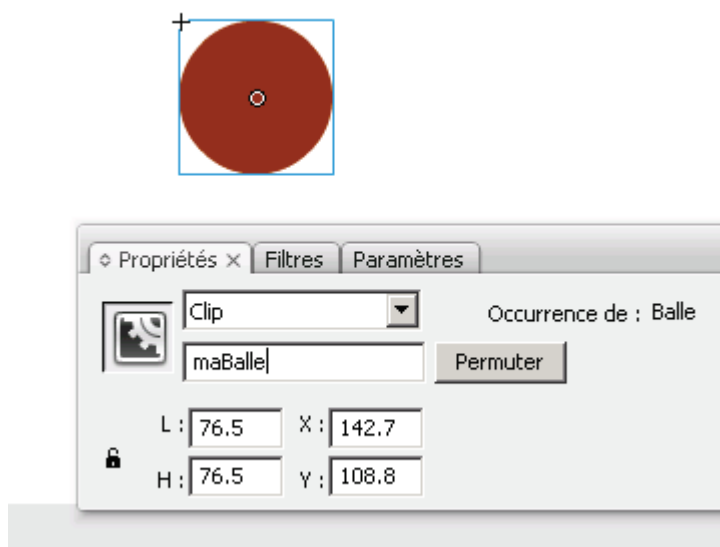
Le symbole clip est sans doute l'objet graphique le plus utilisé dans les animations Flash. Si son fonctionnement n'a pas changé en ActionScript 3, son instanciation et sa manipulation par programmation a été entièrement revue.

Dans un nouveau document Flash CS3 nous créons un symbole de type clip que nous appelons **Balle**. Celui-ci est aussitôt ajouté à la bibliothèque, comme l'illustre la figure 5-2 :



*Figure 5-2. Symbole clip en bibliothèque.*

En posant une occurrence de ce dernier sur la scène nous avons la possibilité de lui affecter à travers l'inspecteur de propriétés un nom d'occurrence comme l'illustre la figure 5-3 :



*Figure 5-3. Occurrence du symbole clip sur la scène principale.*

Aussitôt le nom d'occurrence affecté, Flash ajoutera au scénario concerné une propriété du même nom pointant vers l'occurrence. Le code suivant nous permet d'accéder à cette dernière :

```
| // affiche : [object MovieClip]
| trace( maBalle );
```

Si nous définissons une variable portant le même nom qu'un nom d'occurrence, une erreur à la compilation est générée nous indiquant un conflit de variables.

Cette sécurité évite de définir une variable portant le même nom qu'une occurrence ce qui provoquait avec les précédentes versions d'ActionScript un écrasement de variables difficile à déboguer.

Prenons le cas suivant : au sein d'une animation ActionScript 1 ou 2 un clip posé sur la scène possédait `monMc` comme nom d'occurrence.

Le code suivant retournait une référence vers ce dernier :

```
| // affiche : _level0.monMc
| trace( monMc );
```

Si une variable du même nom était définie, l'accès à notre clip était perdu :

```
| var monMc:String = "bob";
|
| // affiche : bob
| trace( monMc );
```

En ActionScript 3 cette situation ne peut pas se produire grâce à la gestion des conflits de variables à la compilation.

Ici, nous définissons une variable appelée `maBalle` sur le même scénario que notre occurrence :

```
| var maBalle:MovieClip;
```

L'erreur à la compilation suivante est générée :

```
| 1151: Conflit dans la définition maBalle dans l'espace de nom internal.
```

Nous allons nous intéresser à présent à quelques subtilités liées à la propriété `name` de la classe `flash.display.DisplayObject`.

## La propriété `name`

Lors du chapitre 3 intitulé *La liste d'affichage* nous avons vu que la propriété `name` d'un `DisplayObject` pouvait être modifiée dynamiquement.



Une autre nouveauté d'ActionScript 3 intervient dans la manipulation de la propriété `name` des occurrences de symboles placées depuis l'environnement autour de Flash CS3.

En ActionScript 1 et 2 nous pouvions modifier dynamiquement le nom d'un objet graphique grâce à la propriété `_name`. En procédant ainsi nous changions en même temps la variable permettant de le cibler.

Dans l'exemple suivant, un clip possédait `monMc` comme nom d'occurrence :

```
// affiche : monMc
trace( monMc._name );

monMc._name = "monNouveauNom";

// affiche : _level0.monNouveauNom
trace( monNouveauNom );

// affiche : undefined
trace( monMc );
```

En ActionScript 3, cela n'est pas possible pour les occurrences de symboles placées sur la scène manuellement. Seuls les symboles graphiques créés par programmation permettent la modification de leur propriété `name`.

Dans le code suivant nous tentons de modifier la propriété `name` d'une occurrence de symbole posée manuellement :

```
// affiche : maBalle
trace( maBalle.name );

maBalle.name = "monNouveauNom";
```

Ce qui génère l'erreur suivante à l'exécution :

```
Error: Error #2078: Impossible de modifier la propriété de nom d'un objet placé
sur le scénario.
```

La liaison qui existait en ActionScript 1 et 2 entre la propriété `_name` et l'accès à l'occurrence n'est plus valable en ActionScript 3.

Si nous créons un `Sprite` en ActionScript 3, et que nous lui affectons un nom par la propriété `name`, aucune variable du même nom n'est créée pour y accéder :

```
var monSprite:Sprite = new Sprite;

monSprite.name = "monNomOccurrence";

trace( monNomOccurrence );
```

L'erreur à la compilation suivante est générée :

| 1120: Accès à la propriété non définie monNomOccurrence.

Pour y accéder par ce nom, nous utilisons la méthode

`getChildByName` définie par la classe

`flash.display.DisplayObjectContainer`.

```
var monSprite:Sprite = new Sprite;
monSprite.name = "monNomOccurrence";
addChild ( monSprite );
// affiche : [object Sprite]
trace( getChildByName ( "monNomOccurrence" ) );
```

Une autre nouveauté d'ActionScript 3 concerne la suppression des objets graphiques posés depuis l'environnement auteur.

En ActionScript 1 et 2 il était normalement impossible de supprimer de l'affichage ces derniers. Les développeurs devaient utiliser une astuce consistant à passer à une profondeur positive l'objet avant d'appeler une méthode comme `removeMovieClip`.

En ActionScript 3 nous pouvons supprimer tous les objets graphiques posés en dur sur la scène à l'aide de la méthode `removeChild`.

Le code suivant supprime notre occurrence de `Balle` posée manuellement :

```
| removeChild ( maBalle );
```

Alors que les graphistes se contenteront d'animer et manipuler des occurrences de symboles manuellement, un développeur voudra manipuler par programmation les symboles présents au sein de la bibliothèque. Découvrons ensemble le nouveau mécanisme apporté par ActionScript 3.

## A retenir

- Il est impossible de modifier la propriété `name` d'un objet graphique placé manuellement sur la scène.

## Instanciation de symboles par programmation

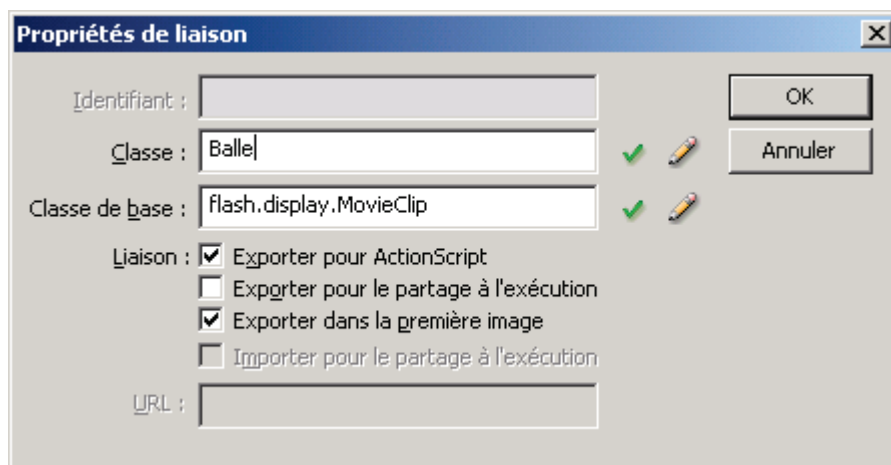
Comme en ActionScript 1 et 2, un symbole clip ne peut être attaché dynamiquement par programmation si l'option *Exporter pour ActionScript* du panneau *Propriétés de liaison* n'est pas activée.

Notons qu'en réalité nous ne donnons plus de nom de liaison au clip en ActionScript 3, mais nous spécifions désormais un nom de classe.

En mode de publication ActionScript 3, le champ *Identifiant* est grisé, nous ne l'utiliserons plus.

En sélectionnant notre symbole *Balle* dans la bibliothèque nous choisissons l'option *Liaison*, puis nous cochons la case *Exporter pour ActionScript*.

La figure 5-4 illustre le panneau *Propriétés de Liaison* :



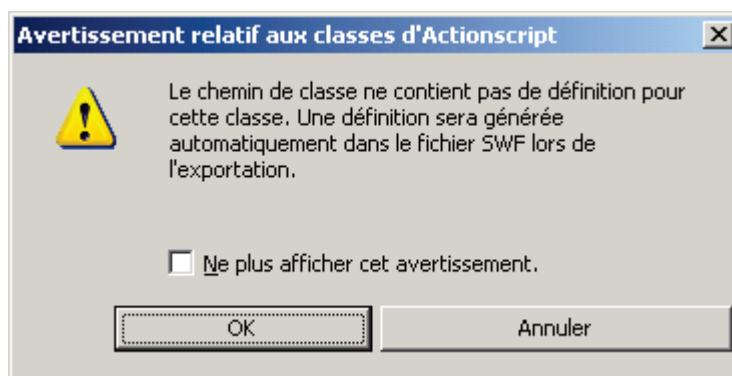
*Figure 5-4. Panneau Propriétés de liaison.*

Le champ *Classe de base* indique la classe dont notre symbole héritera, une fois l'option *Exporter pour ActionScript* cochée, ce dernier est renseigné automatiquement. Notre symbole *Balle* est un clip et hérite donc de la classe `flash.display.MovieClip`.

Le champ *Classe* contient par défaut le nom du symbole en bibliothèque. La grande nouveauté ici, est la possibilité de définir un nom de classe associée au symbole. Nous conservons *Balle*.

En cliquant sur *OK* pour valider, Flash CS3 recherche aussitôt une classe appelée *Balle* à proximité de notre fichier *.fla*. S'il ne trouve aucune classe de ce nom il en génère une automatiquement.

Le panneau de la figure 5-5 nous rapporte cette information :



*Figure 5-5. Panneau de génération automatique de classe.*

Attention, cette classe ne sera pas accessible pour l'édition. Elle sera utilisée en interne par le compilateur pour l'instanciation de notre symbole.

## Instanciation de symboles prédéfinis

En ActionScript 1 et 2 la méthode `attachMovie` permettait d'attacher des symboles par programmation. Dès le départ cette méthode n'était pas très simple à appréhender, ses nombreux paramètres rendaient sa mémorisation difficile. De plus, nous étions obligés d'appeler cette méthode sur une occurrence de `MovieClip`, nous forçant à conserver une référence à un clip pour pouvoir instancier d'autres clips issus de la bibliothèque.

En ActionScript 3 le mot-clé `new` nous permet aussi d'instancier des symboles présents dans la bibliothèque et offre donc beaucoup plus de souplesse que la méthode `attachMovie`.

De n'importe où, sans aucune référence à un `MovieClip` existant, nous pouvons écrire le code suivant pour instancier notre symbole `Balle` :

```
| var maBalle:MovieClip = new Balle();
```

L'utilisation du mot clé `new` pour l'instanciation des objets graphiques et plus particulièrement des symboles résout une autre grande faiblesse d'ActionScript 2 que nous traiterons plus tard dans le chapitre 9 intitulé *Etendre les classes natives*.

Nous avons utilisé le type `MovieClip` pour stocker la référence à notre occurrence de symbole `Balle`, alors que ce symbole possède un type spécifique que nous avons renseigné à travers le panneau *Propriétés de liaison*.

Dans la ligne ci-dessous, nous typons la variable à l'aide du type `Balle` :

```
var maBalle:Balle = new Balle();
```

En testant le type de notre clip `Balle` nous voyons que ce dernier possède plusieurs types communs :

```
var maBalle:Balle = new Balle();

// affiche : true
trace( maBalle is MovieClip );

// affiche : true
trace( maBalle is Balle );
```

Souvenons-nous que notre objet graphique est, pour le moment hors de la liste d'affichage. Pour le voir nous ajoutons notre symbole à notre scénario principal à l'aide de la méthode `addChild` :

```
var maBalle:Balle = new Balle();

addChild ( maBalle );
```

Notre symbole est positionné en coordonnées 0 pour l'axe des x et 0 pour l'axe des y.

Notre instance de la classe `Balle` est donc constituée d'une enveloppe `MovieClip` contenant notre forme vectorielle, ici de forme circulaire. Il paraît donc logique que cette enveloppe contienne un objet graphique enfant de type `flash.display.Shape`.

Le code suivant récupère le premier objet enfant de notre clip `Balle` à l'aide de la méthode `getChildAt` :

```
var maBalle:Balle = new Balle();

addChild (maBalle);

// affiche : [object Shape]
trace( maBalle.getChildAt ( 0 ) );
```

Nous pourrions supprimer le contenu de notre clip avec la méthode `removeChildAt` :

```
var maBalle:Balle = new Balle();

addChild (maBalle);

// affiche : [object Shape]
trace( maBalle.removeChildAt ( 0 ) );
```

Comme nous le découvrons depuis le début de cet ouvrage, ActionScript 3 offre une souplesse sans précédent pour la manipulation des objets graphiques.

Il serait intéressant d'instancier plusieurs objets `Balle` et de les positionner aléatoirement sur la scène avec une taille différente. Pour cela, nous allons au sein d'une boucle `for` créer de multiples instances de la classe `Balle` et les ajouter à la liste d'affichage :

```
var maBalle:Balle;

for ( var i:int = 0; i< 10; i++ )

{

    maBalle = new Balle();

    addChild( maBalle );

}
```

Si nous testons le code précédent nous obtenons dix instances de notre classe `Balle` positionnées en 0,0 sur notre scène.

---

Il s'agit d'un comportement qui a toujours existé dans le lecteur Flash. Tous les objets affichés sont positionnés par défaut en coordonnées 0 pour les axes x et y.

---

Nous allons les positionner aléatoirement sur la scène à l'aide de la méthode `Math.random()`.

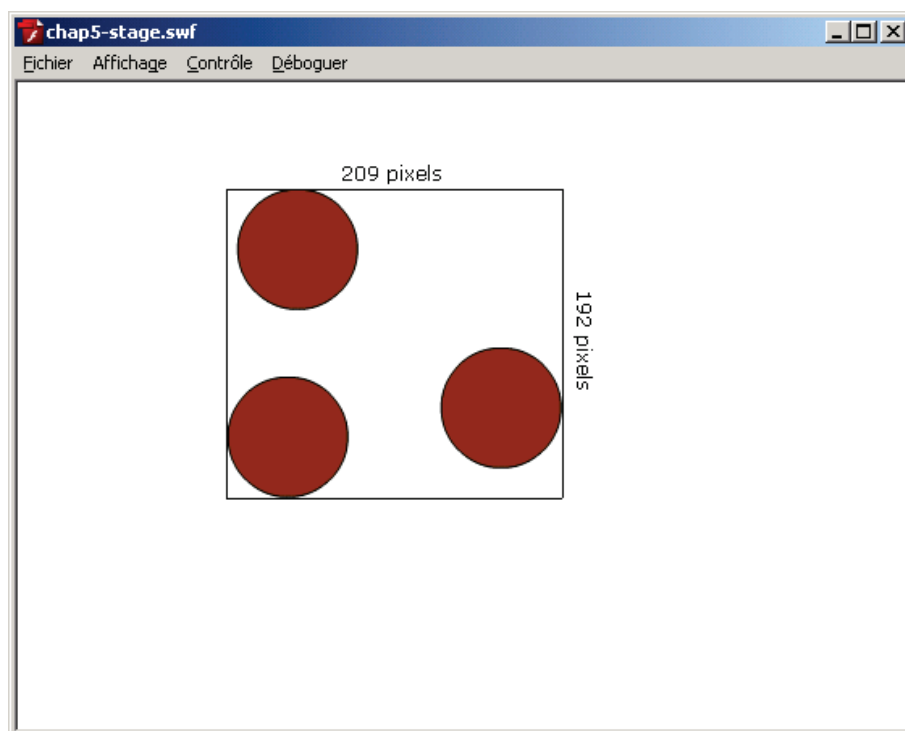
Nous devons donc récupérer la taille totale de la scène pour savoir sur quelle amplitude générer notre valeur aléatoire pour les axes x et y. En ActionScript 1 et 2 nous aurions écrit :

```
maBalle._x = Math.random()*Stage.width;
maBalle._y = Math.random()*Stage.height;
```

En ActionScript 1 et 2, les propriétés `Stage.width` et `Stage.height` nous renvoyaient la taille de la scène. En ActionScript 3 l'objet `Stage` possède quatre propriétés relatives à sa taille, ce qui peut paraître relativement déroutant.

Les deux propriétés `width` et `height` existent toujours mais renvoient la largeur et hauteur occupée par *l'ensemble* des `DisplayObject` contenus par l'objet `Stage`.

La figure 5-6 illustre l'idée :



*Figure 5-6. Comportement des propriétés  
stage.width et stage.height.*

Ainsi dans un SWF vide, les propriétés `stage.width` et `stage.height` renvoient 0.

La figure 5-6 montre une animation où trois instances du symbole `Balle` sont posées sur la scène. Les propriétés `stage.width` et `stage.height` renvoient la surface occupée par les objets graphiques présents dans la liste d’affichage.

Pour récupérer la taille totale de la scène et non la surface occupée par les objets graphiques nous utilisons les propriétés `stage.stageWidth` et `stage.stageHeight` :

```
// affiche : 550
trace( stage.stageWidth );

// affiche : 400
trace( stage.stageHeight );
```

En générant une valeur aléatoire sur la largeur et la hauteur totale nous positionnons aléatoirement nos instances du symbole `Balle` :

```
var maBalle:Balle;

for ( var i:int = 0; i < 10; i++ )
{
```

```

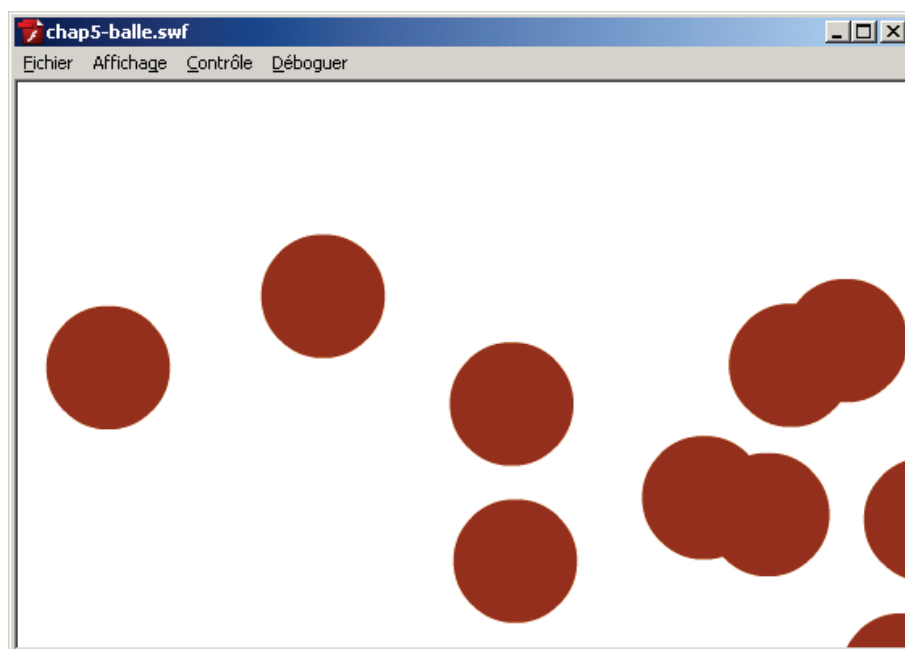
    maBalle = new Balle();

    maBalle.x = Math.random()*stage.stageWidth;
    maBalle.y = Math.random()*stage.stageHeight;

    addChild( maBalle );
}

```

Le code génère l'animation suivante :



*Figure 5-7. Positionnement aléatoire des instances de la classe `Balle`.*

Si nous ne souhaitons pas voir nos occurrences sortir de la scène, nous allons intégrer une contrainte en générant un aléatoire prenant en considération la taille des occurrences.

```

var maBalle:Balle;

for ( var i:int = 0; i< 10; i++ )
{
    maBalle = new Balle();

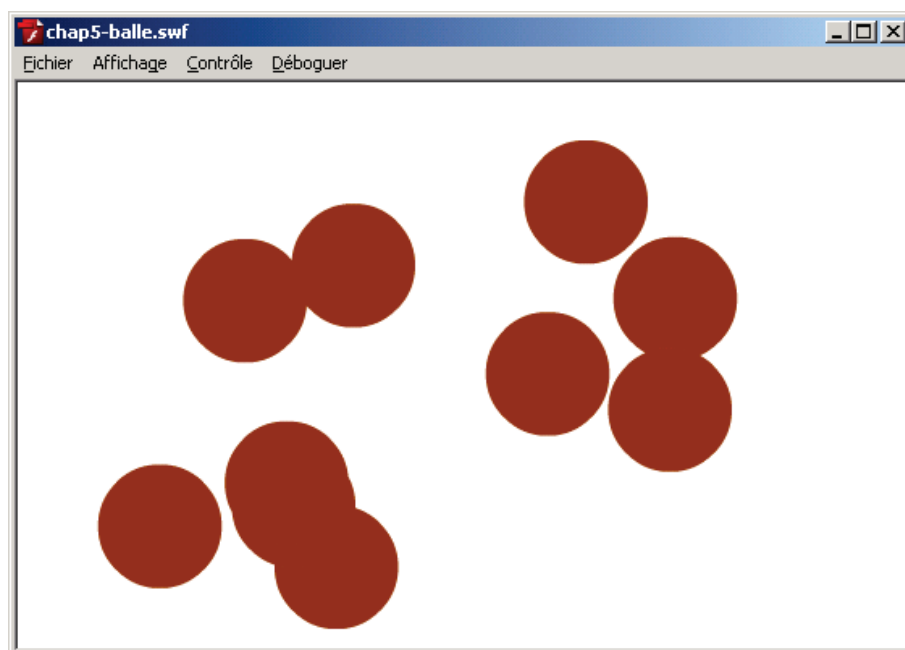
    maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
    maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

    addChild( maBalle );
}

```

La figure 5-8 illustre le résultat :





*Figure 5-8. Positionnement aléatoire sans débordement des instances de la classe `Balle`.*

## A retenir

- Pour instancier un symbole prédéfini, nous utilisons le mot-clé `new`.
- Les propriétés `stage.width` et `stage.height` renvoient la taille occupée par les `DisplayObject` présent au sein de la liste d’affichage.
- Pour récupérer les dimensions de la scène, nous utilisons les propriétés `stage.stageWidth` et `stage.stageHeight`.

## Extraire une classe dynamiquement

Dans les précédentes versions d’ActionScript, nous pouvions stocker les noms de liaisons de symboles au sein d’un tableau, puis boucler sur ce dernier afin d’attacher les objets graphiques :

```
// tableau contenant les identifiants de liaison des symboles
var tableauLiaisons:Array = ["polygone", "balle", "polygone", "carre",
"polygone", "carre", "carre"];

var lng:Number = tableauLiaisons.length;

var ref:MovieClip;

for ( var i:Number = 0; i< lng; i++ )
{
    // affichage des symboles
    ref = this.attachMovie ( tableauLiaisons[i], tableauLiaisons[i] + i, i );
```

```
| }
```

De par l'utilisation du mot-clé `new` afin d'instancier les objets graphiques, nous ne pouvons plus instancier un objet graphique à l'aide d'une simple chaîne de caractères.

Pour réaliser le code équivalent en ActionScript 3, nous devons au préalable extraire une définition de classe, puis instancier cette définition.

Pour cela nous utilisons la fonction

`flash.utils.getDefinitionByName` :

```
// tableau contenant le noms des classes
var tableauLiaisons:Array = ["Polygone", "Balle", "Polygone", "Carre",
"Polygone", "Carre", "Carre"];

var lng:Number = tableauLiaisons.length;

var Reference:Class;

for ( var i:Number = 0; i< lng; i++ )
{

    // extraction des références de classe
    Reference = Class ( getDefinitionByName ( tableauLiaisons[i] ) );

    // instantiation
    var instance:DisplayObject = DisplayObject ( new Reference() );

    // ajout à la liste d'affichage
    addChild ( instance );

}
```

Cette fonctionnalité permet l'évaluation du nom de la classe à extraire de manière dynamique. Nous pourrions imaginer un fichier XML contenant le nom des différentes classes à instancier.

Un simple fichier XML pourrait ainsi décrire toute une interface graphique en spécifiant les objets graphiques devant être instanciés.

## A retenir

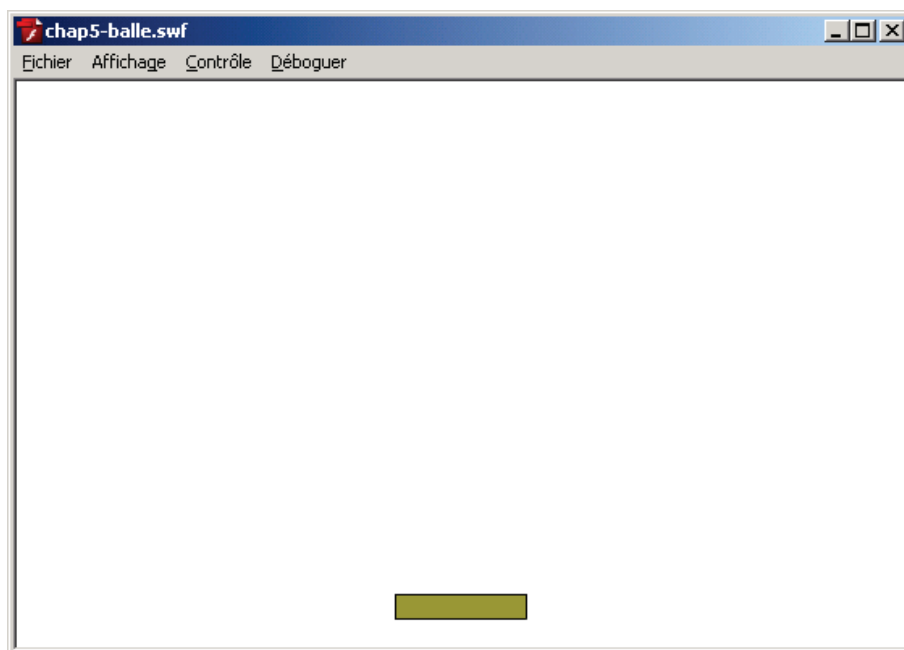
- La fonction `getDefinitionByName` permet d'extraire une définition de classe de manière dynamique.

## Le symbole bouton

Lorsque nous créons un bouton dans l'environnement auteur, celui-ci n'est plus de type `Button` comme en ActionScript 1 et 2 mais de type `flash.display.SimpleButton`.

Il est tout à fait possible de créer et d'enrichir graphiquement des boutons par programmation, chose impossible avec les précédents lecteurs Flash. Nous reviendrons sur cette fonctionnalité dans le prochain chapitre 7 intitulé *Interactivité*.

Nous allons créer un bouton dans le document en cours, puis poser une occurrence de ce dernier sur la scène principale. Dans notre exemple le bouton est de forme rectangulaire, illustrée en figure 5-9 :



*Figure 5-9. Bouton posé sur la scène.*

Nous lui donnons `monBouton` comme nom d'occurrence puis nous testons le code suivant :

```
// affiche : [object SimpleButton]
trace( monBouton );
```

En testant notre animation nous remarquons que notre bouton `monBouton` affiche un curseur représentant un objet cliquable lors du survol. Pour déclencher une action spécifique lorsque nous cliquons dessus, nous utilisons le modèle événementiel que nous avons découvert ensemble au cours du chapitre 3 intitulé *Le modèle événementiel*.

L'événement diffusé par l'objet `SimpleButton` lorsqu'un clic est détecté est l'événement `MouseEvent.CLICK`.

Nous souscrivons une fonction écouteur auprès de ce dernier en ciblant la classe `flash.events.MouseEvent` :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );
```

```
function clicBouton ( pEvt:MouseEvent ):void
{
    // affiche : [object SimpleButton]
    trace( pEvt.target );
}
```

Notre fonction `clicBouton` s'exécute à chaque clic effectué sur notre bouton `monBouton`. La propriété `target` de l'objet événementiel diffusé nous renvoie une référence à l'objet auteur de l'événement, ici notre bouton.

Afin d'attacher dynamiquement nos instances du symbole `Balle` lors du clic sur notre bouton `monBouton`, nous plaçons au sein de la fonction `clicBouton` le processus d'instanciation auparavant créé :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

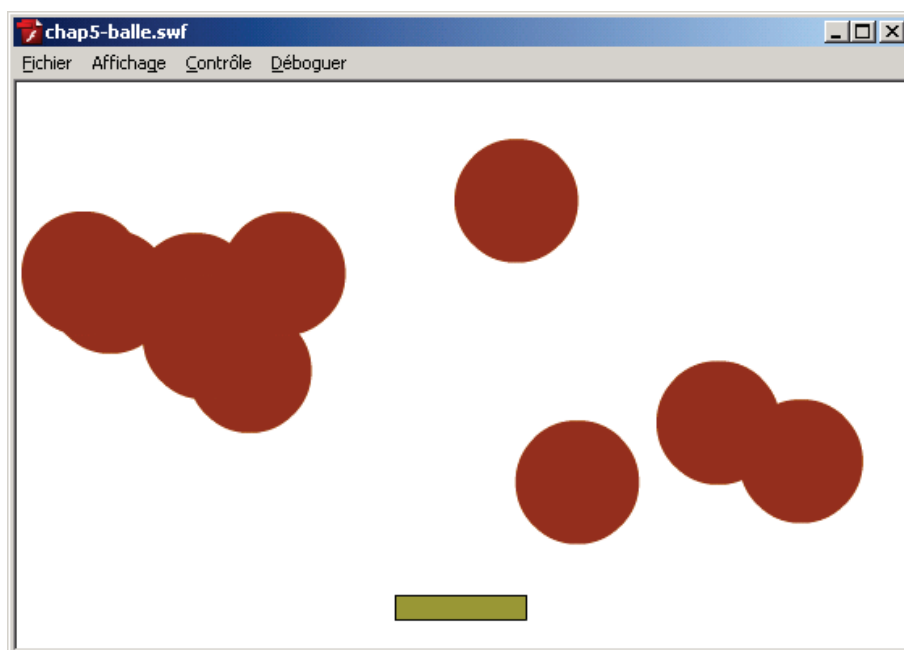
function clicBouton ( pEvt:MouseEvent ):void
{
    var maBalle:Balle

    for ( var i:int = 0; i< 10; i++ )
    {
        maBalle = new Balle();

        maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
        maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

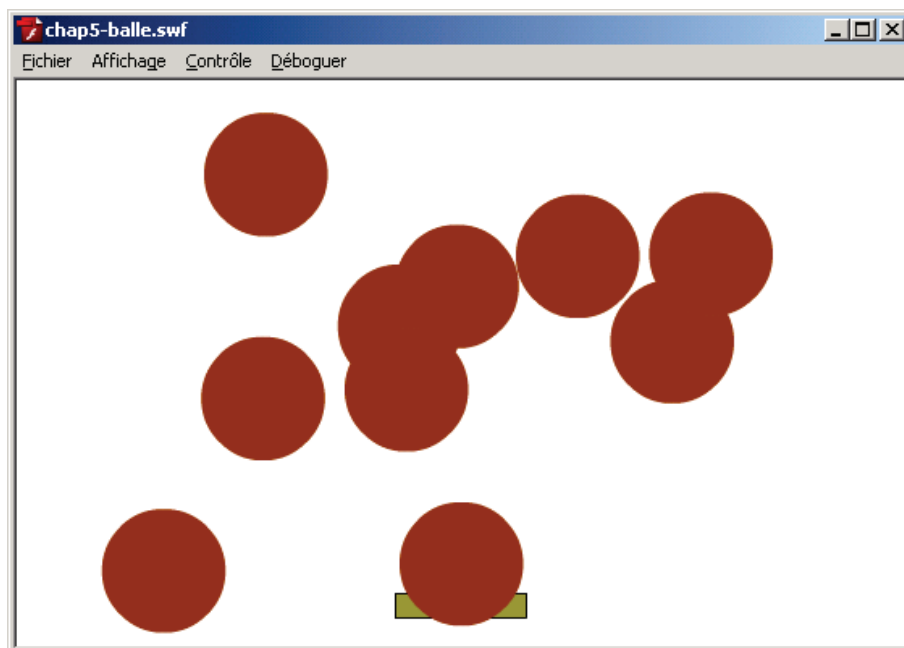
        addChild ( maBalle );
    }
}
```

Au clic souris sur notre bouton, dix instances du symbole `Balle` sont attachées sur le scénario principal. Comme l'illustre la figure 5-10 :



*Figure 5-10. Occurrences de symbole `Balle`.*

Nous risquons d’être confrontés à un problème de chevauchement, il serait intéressant de remonter notre bouton au premier niveau pour éviter qu’il ne soit masqué par les instances de la classe `Balle`, comme l’illustre la figure 5-11 :



*Figure 5-11. Bouton masqué par les instances de la classe `Balle`.*

Pour ramener un objet graphique au premier plan nous devons travailler sur son index au sein de la liste d’affichage. Nous savons que la propriété `numChildren` nous renvoie le nombre d’enfants total, `numChildren-1` correspond donc à l’index de l’objet le plus haut de la pile.

En échangeant l’index de notre bouton et du dernier objet enfant avec la méthode `setChildIndex` nous obtenons le résultat escompté :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{
    var maBalle:Balle

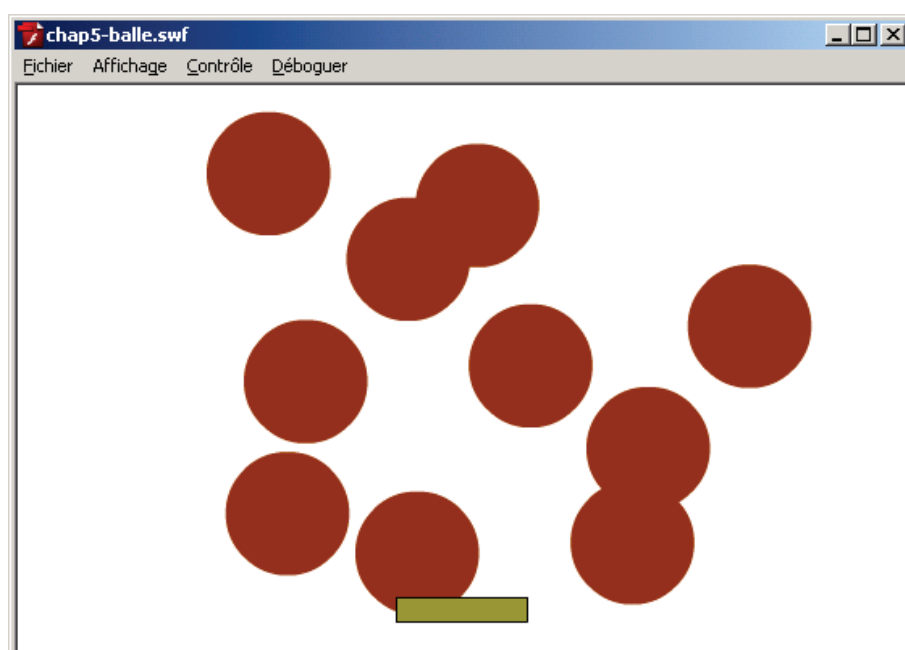
    for ( var i:int = 0; i< 10; i++ )
    {
        maBalle = new Balle();

        maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
        maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

        addChild( maBalle );
    }

    setChildIndex ( monBouton, numChildren - 1 );
}
```

La figure 5-12 illustre le résultat :



*Figure 5-12. Bouton placé au premier plan.*

La méthode `addChild` empile chaque instance sans jamais supprimer les objets graphiques déjà présents sur la scène.

Pour supprimer tous les objets graphiques nous pourrions parcourir la scène en supprimant chaque occurrence de type `Balle`.

L'idée serait d'ajouter une fonction `nettoie` avant la boucle `for`, afin de supprimer les occurrences du symbole `Balle` déjà présentes sur la scène :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

function nettoie ( pConteneur:DisplayObjectContainer, pClasse:Class ):void
{
    var monDisplayObject:DisplayObject;

    for ( var i:int = pConteneur.numChildren-1; i >= 0; i-- )
    {
        monDisplayObject = pConteneur.getChildAt ( i );

        if ( monDisplayObject is pClasse ) pConteneur.removeChild
(monDisplayObject);
    }
}

function clicBouton ( pEvt:MouseEvent ):void
{
    nettoie ( this, Balle );

    var maBalle:Balle;

    for ( var i:int = 0; i < 10; i++ )
    {
        maBalle = new Balle();

        maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
        maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

        addChild( maBalle );
    }

    setChildIndex ( monBouton, numChildren - 1 );
}
```

La fonction `nettoie` parcourt le conteneur passé en paramètre ici notre scène, puis récupère chaque objet enfant et stocke sa référence

dans une variable `monDisplayObject` de type `flash.display.DisplayObject`.

Nous utilisons ce type pour notre variable `monDisplayObject` car celle-ci peut être amenée à référencer différents sous-types de la classe `DisplayObject`. Nous choisissons ce type qui est le type commun à tout objet enfant.

Nous testons son type à l'aide du mot-clé `is`. Si celui-ci est de type `Balle` alors nous supprimons l'instance de la liste d'affichage.

Nous pourrions obtenir une structure plus simple en créant un objet graphique conteneur afin d'accueillir les occurrences du symbole `Balle`, puis vider entièrement ce dernier sans faire de test.

De plus si nous souhaitons déplacer tous les objets plus tard, cette approche s'avèrera plus judicieuse :

```
monBouton.addEventListener( MouseEvent.CLICK, clicBouton );

var conteneur:Sprite = new Sprite();
addChildAt ( conteneur, 0 );

function nettoie ( pConteneur:DisplayObjectContainer ):void
{
    while ( pConteneur.numChildren ) pConteneur.removeChildAt ( 0 );
}

function clicBouton ( pEvt:MouseEvent ):void
{
    nettoie ( conteneur );

    var maBalle:Balle;

    for ( var i:int = 0; i< 10; i++ )
    {
        maBalle = new Balle();

        maBalle.x = Math.random()*(stage.stageWidth - maBalle.width);
        maBalle.y = Math.random()*(stage.stageHeight - maBalle.height);

        conteneur.addChild( maBalle );
    }
}
```

La fonction `nettoie` intègre une boucle `while` supprimant chaque objet enfant, tant qu'il en existe. En cliquant sur notre bouton nous



nettoyons le conteneur de type `flash.display.Sprite` puis nous y ajoutons nos occurrences du symbole `Balle`.

Revenons sur certaines parties du code précédent, dans un premier temps nous créons un conteneur de type `flash.display.Sprite`. Nous utilisons un objet `Sprite` car utiliser un `MovieClip` ici ne serait d'aucune utilité, un objet `Sprite` suffit car nous devons simplement y stocker des objets enfants :

```
var conteneur:Sprite = new Sprite;  
  
addChildAt ( conteneur, 0 );
```

Nous créons un conteneur puis nous l'ajoutons à la liste d'affichage à l'aide de la méthode `addChildAt` en le plaçant à l'index 0 déjà occupé par notre bouton seul objet graphique présent à ce moment là.

Notre conteneur prend donc l'index du bouton déplaçant ce dernier à l'index 1. Notre bouton se retrouve ainsi au-dessus de l'objet conteneur, évitant ainsi d'être caché par les instances de la classe `Balle`.

La fonction `nettoie` prend en paramètre le conteneur à nettoyer, et supprime chaque enfant tant que celui-ci en possède :

```
function nettoie ( pConteneur:DisplayObjectContainer ):void  
{  
    while ( pConteneur.numChildren ) pConteneur.removeChildAt ( 0 );  
}
```

Cette fonction `nettoie` peut être réutilisée pour supprimer tous les objets enfants de n'importe quel `DisplayObjectContainer`.

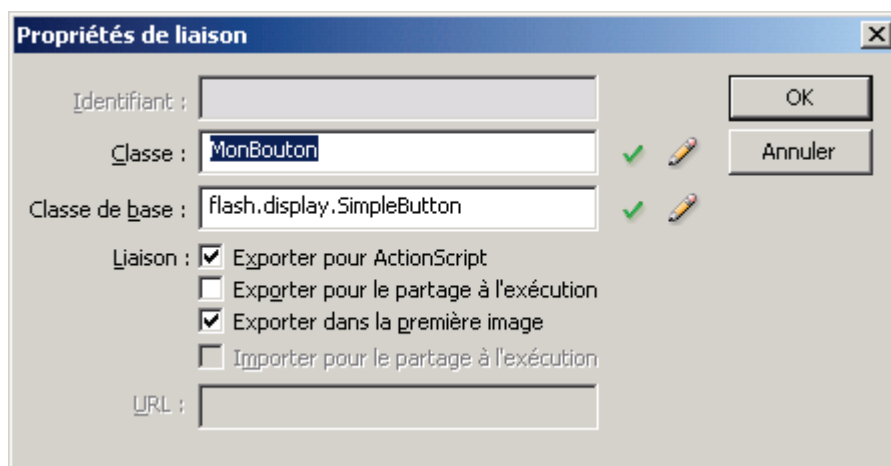
## A retenir

- En ActionScript 3, il est recommandé de créer des conteneurs afin de manipuler plus facilement un ensemble d'objets graphiques.

## Le symbole bouton

Notre symbole bouton en bibliothèque peut aussi être attaché dynamiquement à un scénario en lui associant une classe spécifique grâce au panneau de liaison.

En sélectionnant l'option *Liaison* sur notre symbole `Bouton` nous faisons apparaître le panneau *Propriétés de liaison*.



*Figure 5-13. Propriétés de liaison d'un symbole de type Bouton.*

En cochant l'option *Exporter pour ActionScript* nous rendons notre symbole disponible par programmation. Nous spécifions *MonBouton* comme nom de classe, puis nous cliquons sur *OK*.

Pour instancier notre bouton dynamiquement nous écrivons :

```
var monBouton:MonBouton = new MonBouton();
addChild ( monBouton );
```

Souvenez-vous, il était impossible en ActionScript 1 et 2, de créer de véritables boutons par programmation. Les développeurs devaient obligatoirement utiliser des clips ayant un comportement bouton.

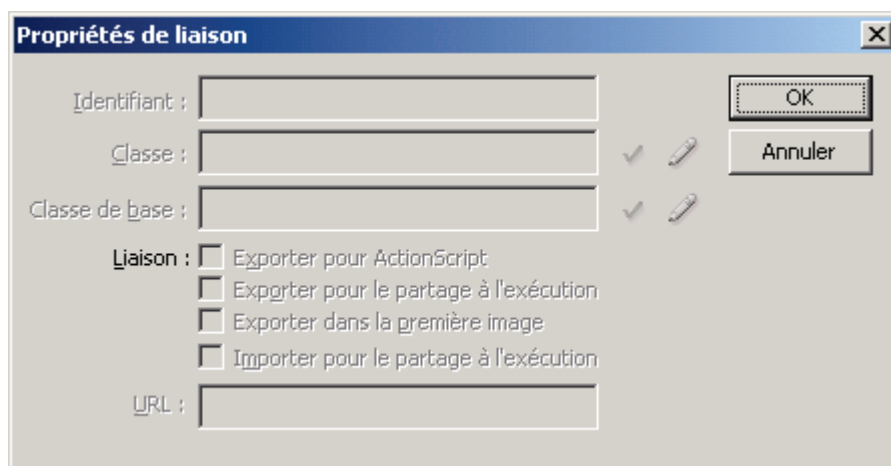
Nous verrons au cours du chapitre 7 intitulé *Interactivité* que l'utilisation de la classe *SimpleButton* s'avère en réalité rigide et n'offre pas une souplesse équivalente à la classe *MovieClip*.

## Le symbole graphique

Lorsqu'un symbole graphique est placé en bibliothèque, celui-ci ne peut être manipulé par programmation et instancié dynamiquement au sein de l'application. De par sa non-interactivité ce dernier est de type *flash.display.Shape*.

Pour rappel, la classe *Shape* n'hérite pas de la classe *flash.display.interactiveObject* et ne peut donc avoir une quelconque interactivité liée au clavier ou la souris.

Lorsque nous sélectionnons l'option *Liaison* sur un symbole graphique toutes les options sont grisées, comme le démontre la figure 5-14 :



*Figure 5-14. Options de liaisons grisées pour le symbole graphique.*

Il est cependant possible de manipuler un graphique posé sur la scène en y accédant grâce aux méthodes d'accès de la liste d'affichage comme `getChildAt` comme nous l'avons vu durant le chapitre 4.

Nous sommes obligés de pointer notre objet graphique en passant un index car aucun nom d'occurrence ne peut être affecté à une occurrence de symbole graphique.

Si nous posons une occurrence de graphique sur une scène vide nous pouvons tout de même y accéder par programmation à l'aide du code suivant :

```
var monGraphique:DisplayObject = getChildAt ( 0 );  
  
// affiche : [object Shape]  
trace( monGraphique );
```

Nous pourrions être tentés de lui affecter un nom par la propriété `name`, et d'y faire référence à l'aide de la méthode `getChildByName` définie par la classe `flash.display.DisplayObjectContainer`.

Malheureusement, comme nous l'avons vu précédemment, la modification de la propriété `name` d'un objet graphique posé depuis l'environnement auteur est impossible.

Notez que si dans l'environnement auteur de Flash nous créons un graphique et imbriquons à l'intérieur un clip, à l'exécution notre imbrication ne pourra être conservée.

En regardant de plus près l'héritage de la classe `Shape` nous remarquons que celle-ci n'hérite pas de la classe `flash.display.DisplayObjectContainer` et ne peut donc contenir des objets enfants.

Il peut pourtant arriver qu'un clip soit imbriqué depuis l'environnement auteur dans un graphique, même si dans l'environnement auteur cela ne pose aucun problème, cette imbrication ne pourra être conservée à l'exécution.

Le clip sortira du graphique pour devenir un enfant direct du parent du graphique. Les deux objets se retrouvant ainsi au même niveau d'imbrication, ce comportement bien que troublant s'avère tout à fait logique et doit être connu afin qu'il ne soit pas source d'interrogations.

## A retenir

- Le symbole graphique ne peut pas être associé à une sous classe.

## Les images bitmap

Depuis toujours nous pouvons intégrer au sein de sa bibliothèque des images bitmap de différents types. En réalité, il faut considérer dans Flash une image bitmap comme un symbole.

Depuis Flash 8 nous avons la possibilité d'attribuer un nom de liaison à une image et de l'attacher dynamiquement à l'aide de la méthode `BitmapData.loadBitmap` puis de l'afficher à l'aide de la méthode `MovieClip.attachBitmap`.

Pour attacher une image de la bibliothèque nous devons appeler la méthode `loadBitmap` de la classe `BitmapData` :

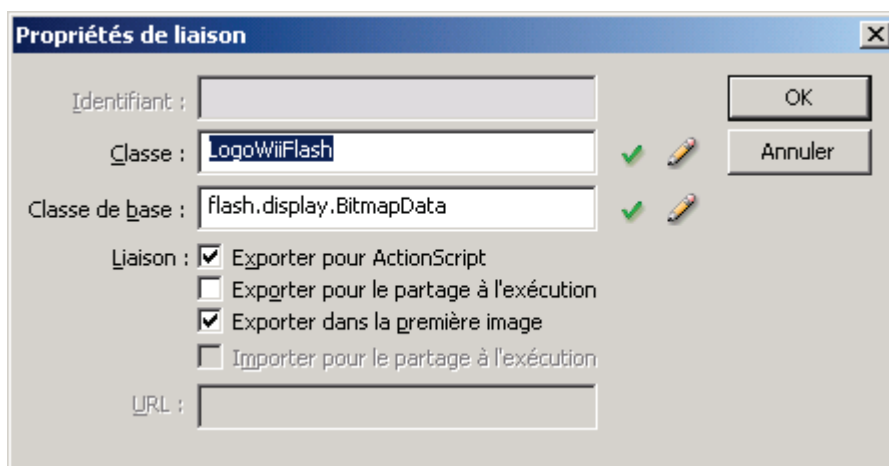
```
import flash.display.BitmapData;
var ref:BitmapData = BitmapData.loadBitmap ("LogoWiiFlash");
```

En lui passant un nom d'identifiant affecté par le panneau liaison, nous récupérons notre image sous la forme de `BitmapData` puis nous l'affichions avec la méthode `attachBitmap` :

```
import flash.display.BitmapData;
var monBitmap:BitmapData = BitmapData.loadBitmap ("LogoWiiFlash");
attachBitmap ( monBitmap, 0 );
```

En ActionScript 3, les images s'instancient comme tout objet graphique avec le mot-clé `new`.

Importez une image de type quelconque dans la bibliothèque puis faites un clic-droit sur celle-ci pour sélectionner l'option *Liaison* comme illustrée en figure 5-15 :



*Figure 5-15. Propriétés de liaison d'un symbole de type bitmap.*

En cochant l'option *Exporter pour ActionScript* nous rendons les champs *Classe* et *Classe de base* éditables.

Nous spécifions `LogoWiiFlash` comme nom de classe, et laissons comme classe de base `flash.display.BitmapData`. Notre image sera donc de type `LogoWiiFlash` héritant de `BitmapData`.

Puis nousinstancions notre image :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash();
```

Si nous testons le code précédent, le message d'erreur suivant s'affiche :

```
1136: Nombre d'arguments incorrect. 2 attendus.
```

En ActionScript 3, un objet `BitmapData` ne peut être instancié sans préciser une largeur et hauteur spécifique au constructeur. Afin de ne pas être bloqué à la compilation nous devons obligatoirement passer une largeur et hauteur de 0,0 :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash(0,0);
```

A l'exécution, le lecteur affiche l'image à sa taille d'origine.

## A retenir

- Afin d’instancier une image bitmap issue de la bibliothèque, nous devons *obligatoirement* spécifier une hauteur et une largeur de 0 pixels. A l’exécution, le lecteur affiche l’image à sa taille d’origine.

## Afficher une image bitmap

En ActionScript 1 et 2, une image `BitmapData` pouvait être affichée en étant enveloppée dans un `MovieClip` grâce à la méthode `MovieClip.attachBitmap`.

En ActionScript 3 si nous tentons d’ajouter à la liste d’affichage un objet graphique de type `BitmapData` nous obtenons l’erreur suivante à la compilation :

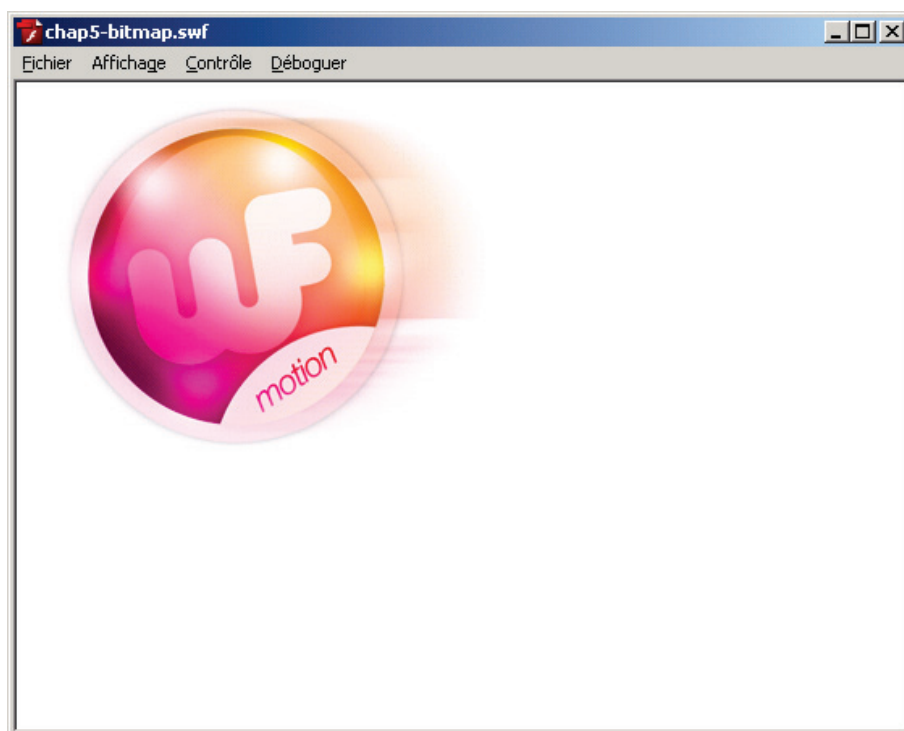
```
1067: Contrainte implicite d'une valeur du type flash.display:BitmapData vers un type sans rapport flash.display:DisplayObject.
```

Pour ajouter à la liste d’affichage un `BitmapData` nous devons *obligatoirement* l’envelopper dans un objet de type `Bitmap`. Celui-ci nous permettra plus tard de positionner l’objet `BitmapData` et d’effectuer d’autres manipulations sur ce dernier.

Nousinstancions un objet `Bitmap`, puis nous passons au sein de son constructeur l’objet `BitmapData` à envelopper :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash(0, 0);  
var monBitmap:Bitmap = new Bitmap( monBitmapData );  
addChild ( monBitmap );
```

Une fois compilé, notre image est bien affichée :



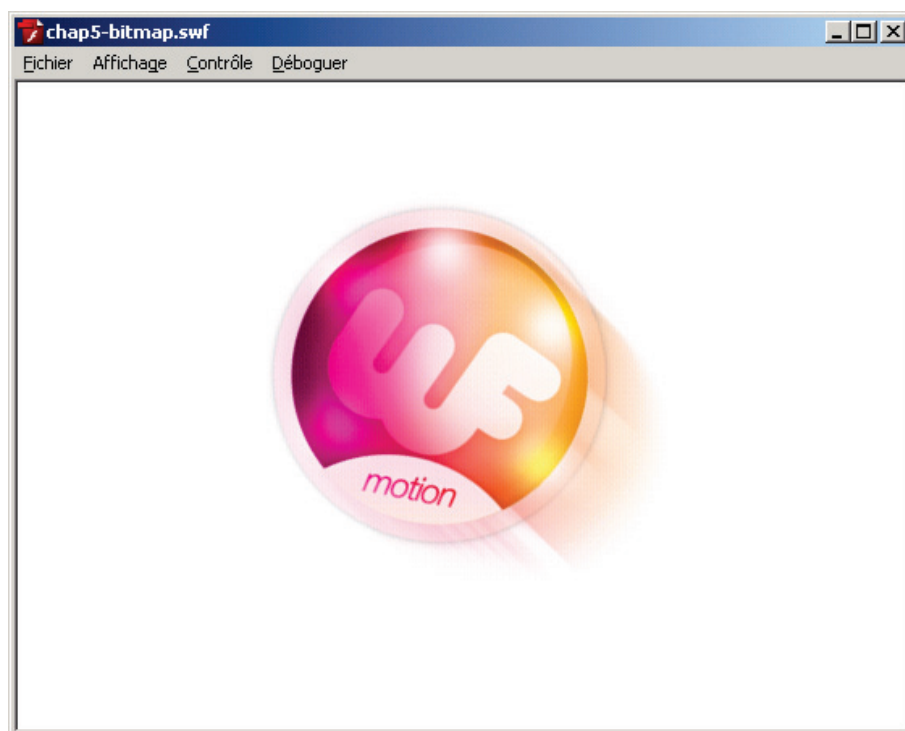
*Figure 5-16. Image bitmap ajoutée à la liste d'affichage.*

La classe `flash.display.Bitmap` héritant de la classe `flash.display.DisplayObject` possède toutes les propriétés et méthodes nécessaires à la manipulation d'un objet graphique.

Pour procéder à une translation de notre image, puis une rotation nous pouvons écrire :

```
var monBitmapData:LogoWiiFlash = new LogoWiiFlash(0, 0);  
  
var monBitmap:Bitmap = new Bitmap( monBitmapData );  
  
addChild ( monBitmap );  
  
monBitmap.smoothing = true;  
monBitmap.rotation = 45;  
monBitmap.x += 250;
```

Le code suivant, génère le rendu suivant :



*Figure 5-17. Translation et rotation sur un objet  
Bitmap.*

Nous reviendrons en détail sur la classe `Bitmap` au cours du chapitre 12 intitulé *Programmation bitmap*.

## A retenir

- Un symbole de type graphique n'est pas manipulable depuis la bibliothèque par programmation.
- Une image est associée au type `flash.display.BitmapData`, pour afficher celle-ci nous devons l'envelopper dans un objet `flash.display.Bitmap`.

## Le symbole Sprite

L'objet graphique `Sprite` est très proche du classique `MovieClip` et possède quasiment toutes ses fonctionnalités mais ne contient pas de scénario et donc aucune des méthodes liées à sa manipulation telles `gotoAndStop`, `gotoAndPlay`, etc.

C'est en quelque sorte un `MovieClip` version allégée.

Le développeur Flash qui a toujours eu l'habitude de créer des clips vides dynamiquement se dirigera désormais vers l'objet `Sprite` plus léger en mémoire et donc plus optimisé.

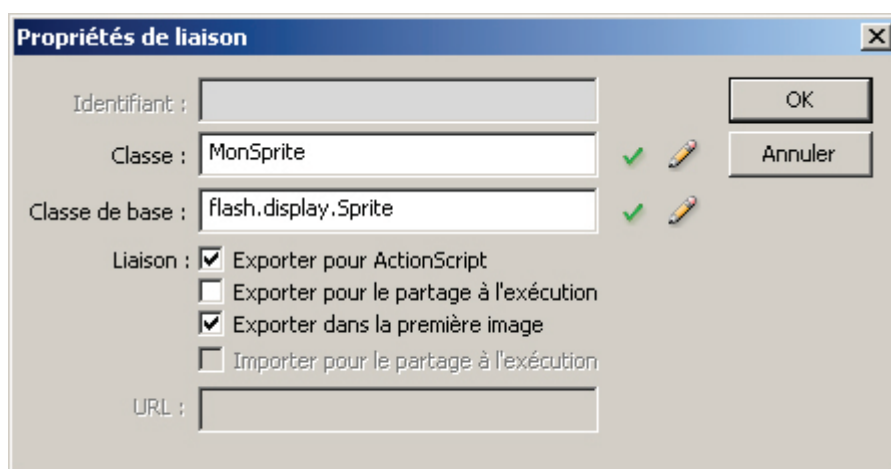


Lorsque nous transformons une forme en symbole l'environnement de Flash CS3 nous propose trois types de symboles :

- Clip
- Bouton
- Graphique

Le type `Sprite` n'apparaît pas, mais il est pourtant possible de créer un symbole de type `Sprite` depuis l'environnement auteur de Flash CS3. Pour cela, nous allons ouvrir un nouveau document et créer un symbole de type clip. Dans le panneau *Propriétés de liaison*, nous cochons la case *Exporter pour ActionScript*.

Nous définissons comme nom de classe `MonSprite`, dans le champ *Classe de base* nous remplaçons la classe `flash.display.MovieClip` par la classe `flash.display.Sprite`, comme l'illustre la figure 5-18 :



*Figure 5-18. Panneau de propriétés de liaison.*

De cette manière notre symbole héritera de la classe `Sprite` au lieu de la classe `MovieClip`.

Nousinstancions notre symbole :

```
var monSprite:MonSprite = new MonSprite();
```

Puis nous l'affichons :

```
var monSprite:MonSprite = new MonSprite();

monSprite.x = 200;
monSprite.y = 100;

addChild ( monSprite );
```

Si nous testons son type avec l'opérateur `is`, nous voyons que notre occurrence est bien de type `Sprite` :

```
var monSprite:MonSprite = new MonSprite();

monSprite.x = 200;
monSprite.y = 100;

addChild ( monSprite );

// affiche : true
trace( monSprite is Sprite );
```

Au sein de l'environnement de Flash CS3, le symbole `Sprite` est similaire au symbole clip mais l'absence de scénario n'est pas répercutée graphiquement.

Comme nous l'avons vu précédemment, l'objet graphique `Sprite` n'a pas de scénario, mais que se passe t-il si nous ajoutons une animation au sein de ce dernier ?

A l'exécution le symbole ne lira pas l'animation, si nous passons la classe de base du symbole à `flash.display.MovieClip` l'animation est alors jouée.

### A retenir

- Même si le symbole de type `flash.display.Sprite` n'est pas disponible depuis le panneau *Convertir en symbole* il est possible de créer des symboles `Sprite` depuis l'environnement auteur de Flash CS3.
- Pour cela, nous utilisons le champ *Classe de base* en spécifiant la classe `flash.display.Sprite`.
- Si une animation est placée au sein d'un symbole de type `Sprite`, celle-ci n'est pas jouée.

## Définition du code dans un symbole

Dans les précédentes versions d'ActionScript, la définition de code était possible sur les occurrences de la manière suivante :

```
on (release)
{
    trace("cliqué");
}
```

Même si cette technique était déconseillée dans de larges projets, elle permettait néanmoins d'attacher simplement le code aux objets. En ActionScript 3, la définition de code est impossible sur les

occurrences, mais il est possible de reproduire le même comportement en ActionScript 3.

En plaçant le code suivant sur le scénario d'un `MovieClip` nous retrouvons le même comportement :

```
// écoute de l'événement MouseEvent.CLICK
addEventListener ( MouseEvent.CLICK, clic );

// activation du comportement bouton
buttonMode = true;

function clic ( pEvt:MouseEvent ):void
{
    trace("cliqué");
}
```

Nous reviendrons sur la propriété `buttonMode` au cours du chapitre 7 intitulé *Interactivité*.

# 6

## Propagation événementielle

<b>CONCEPT .....</b>	<b>1</b>
<b>LA PHASE DE CAPTURE .....</b>	<b>3</b>
LA NOTION DE NŒUDS.....	6
DETERMINER LA PHASE EN COURS .....	7
<b>OPTIMISER LE CODE AVEC LA PHASE DE CAPTURE .....</b>	<b>9</b>
<b>LA PHASE CIBLE .....</b>	<b>14</b>
<b>INTERVENIR SUR LA PROPAGATION.....</b>	<b>18</b>
<b>LA PHASE DE REMONTEE .....</b>	<b>23</b>
<b>ECOUTER PLUSIEURS PHASES .....</b>	<b>27</b>
LA PROPRIETE EVENT.BUBBLES .....	29
<b>SUPPRESSION D'ECOUTEURS .....</b>	<b>30</b>

### Concept

Nous avons abordé la notion d'événements au cours du chapitre 3 intitulé *Modèle événementiel* et traité les nouveautés apportées par ActionScript 3. La propagation événementielle est un concept avancé d'ActionScript 3 qui nécessite une attention toute particulière. Nous allons au cours de ce chapitre apprendre à maîtriser ce nouveau comportement.

Le concept de propagation événementielle est hérité du *Document Object Model (DOM)* du W3C dont la dernière spécification est disponible à l'adresse suivante :

<http://www.w3.org/TR/DOM-Level-3-Events/>

En ActionScript 3, seuls les objets graphiques sont concernés par la propagation événementielle. Ainsi, lorsqu'un événement survient au sein de la liste d'affichage, le lecteur Flash propage ce dernier de l'objet `flash.display.Stage` jusqu'au parent direct de l'objet auteur de la propagation.

Cette descente de l'événement est appelée *phase de capture*.

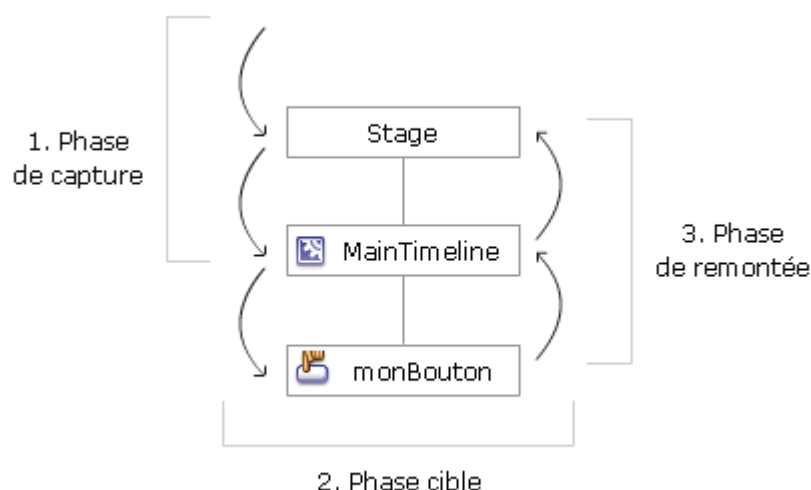
Grâce à ce mécanisme, nous pouvons souscrire un écouteur auprès de l'un des parents de l'objet ayant initié la propagation, afin d'en être notifié durant la phase descendante. Le terme de capture est utilisé car nous pouvons *capturer* l'événement durant sa descente et stopper sa propagation si nécessaire.

*La phase cible* intervient lorsque l'événement a parcouru tous les objets parents et atteint l'objet ayant provoqué sa propagation, ce dernier est appelé *objet cible* ou *nœud cible*.

Une fois la phase cible terminée, le lecteur Flash propage l'événement dans le sens inverse de la phase de capture. Cette phase est appelée *phase de remontée*.

Durant celle-ci l'événement remonte jusqu'à l'objet `Stage` c'est-à-dire à la racine de la liste d'affichage.

Ces trois phases constituent *la propagation événementielle*, schématisée sur la figure 6-1 :



*Figure 6-1. Les trois phases du flot événementiel.*

Les événements diffusés en ActionScript 2 ne disposaient que d'une seule phase cible. Les deux nouvelles phases apportées par

ActionScript 3 vont nous permettre de mieux gérer l'interaction entre les objets graphiques au sein de la liste d'affichage.

---

Attention, la propagation événementielle ne concerne que les objets graphiques.

---

Il est important de noter qu'ActionScript 3 n'est pas le seul langage à intégrer la notion de propagation événementielle, des langages comme JavaScript ou Java, intègrent ce mécanisme depuis plusieurs années déjà.

Nous allons nous attarder sur chacune des phases, et découvrir ensemble quels sont les avantages liés à cette propagation événementielle et comment en tirer profit dans nos applications.

### A retenir

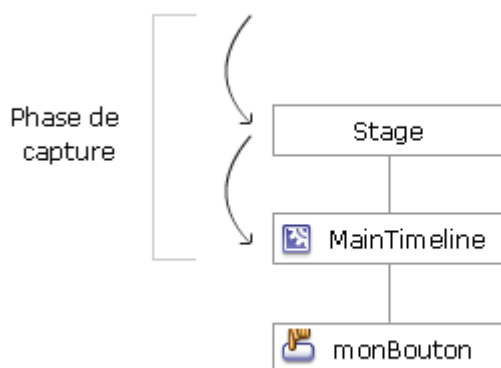
- La propagation événementielle ne concerne que les objets graphiques.
- La propagation événementielle n'est pas propre à ActionScript 3. D'autres langages comme JavaScript, Java ou C# l'intègre depuis plusieurs années déjà.
- Le flot événementiel se divise en trois phases distinctes : la phase de capture, la phase cible, et la phase de remontée.

## La phase de capture

Comme nous l'avons abordé précédemment, la phase de capture est la première phase du flot événementiel.

En réalité, lorsque nous cliquons sur un bouton présent au sein de la liste d'affichage, tous les objets graphiques parents à ce dernier sont d'abord notifiés de l'événement et peuvent ainsi le diffuser.

La figure 6-2 illustre une situation classique, un bouton est posé sur le scénario principal :



*Figure 6-2. Phase de capture.*

Lorsque l'utilisateur clique sur le bouton `monBouton`, la phase de capture démarre. Le lecteur Flash propage l'événement du haut vers le bas de la hiérarchie.

---

Notons que la descente de l'événement s'arrête au parent direct du bouton, la phase suivante sera la phase cible.

---

L'objet `Stage`, puis l'objet `MainTimeline` (`root`) sont ainsi notifiés de l'événement. Ainsi, si nous écoutons l'événement en cours de propagation sur l'un de ces derniers, il nous est possible de l'intercepter.

En lisant la signature de la méthode `addEventListener` nous remarquons la présence d'un paramètre appelé `useCapture` :

```
public function addEventListener(type:String, listener:Function,
useCapture:Boolean = false, priority:int = 0, useWeakReference:Boolean =
false):void
```

Par défaut la méthode `addEventListener` ne souscrit pas l'écouteur passé à la phase de capture. Pour en profiter nous devons passer la valeur booléenne `true` au troisième paramètre nommé `useCapture`.

L'idée est d'écouter l'événement sur l'un des objets parents à l'objet cible :

```
| objetParent.addEventListener ( MouseEvent.CLICK, clicBouton, true );
```

Passons à un peu de pratique, dans un nouveau document Flash CS3 nous créons un symbole de type bouton et nous posons une occurrence de ce dernier, appelée `monBouton`, sur la scène principale.

Dans le code suivant nous souscrivons un écouteur auprès du scénario principal en activant la capture :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
{
    // affiche : [MouseEvent type="click" bubbles=true cancelable=false
    eventPhase=1 localX=13 localY=13 stageX=75.95 stageY=92 relatedObject=null
    ctrlKey=false altKey=false shiftKey=false delta=0]
    trace( pEvt );
}
```

Lors du clic sur notre bouton `monBouton`, l'événement `MouseEvent.CLICK` entame sa phase de descente puis atteint le scénario principal qui le diffuse aussitôt. La fonction écouteur `clicBouton` est déclenchée.

Contrairement à la méthode `hasEventListener`, la méthode `willTrigger` permet de déterminer si un événement spécifique est écouté auprès de l'un des parents lors de la phase de capture ou de remontée :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
{
    trace( pEvt );
}

// aucun écouteur n'est enregistré auprès du bouton
// affiche : false
trace( monBouton.hasEventListener (MouseEvent.CLICK) );

// un écouteur est enregistré auprès d'un des parents du bouton
// affiche : true
trace( monBouton.willTrigger( MouseEvent.CLICK ) );

// un écouteur est enregistré auprès du scénario principal
// affiche : true
trace( willTrigger( MouseEvent.CLICK ) );
```

Nous préférons donc l'utilisation de la méthode `willTrigger` dans un contexte de propagation événementielle.

Au cours du chapitre 3 intitulé *Modèle événementiel* nous avons découvert la propriété `target` de l'objet événementiel correspondant à l'objet auteur du flot événementiel que nous appelons généralement *objet cible*.

Attention, durant la phase de capture ce dernier n'est pas celui qui a diffusé l'événement, mais celui qui est à la *source de la propagation* événementielle. Dans notre cas, l'objet `MainTimeline` a diffusé



l'événement `MouseEvent.CLICK`, suite à un déclenchement de la propagation de l'événement par notre bouton.

L'intérêt majeur de la phase de capture réside donc dans la possibilité d'intercepter depuis un parent n'importe quel événement provenant d'un enfant.

### La notion de nœuds

Lorsqu'un événement se propage, les objets graphiques notifiés sont appelés *objets nœuds*. Pour récupérer le nœud sur lequel se trouve l'événement au cours de sa propagation, nous utilisons la propriété `currentTarget` de l'objet événementiel.

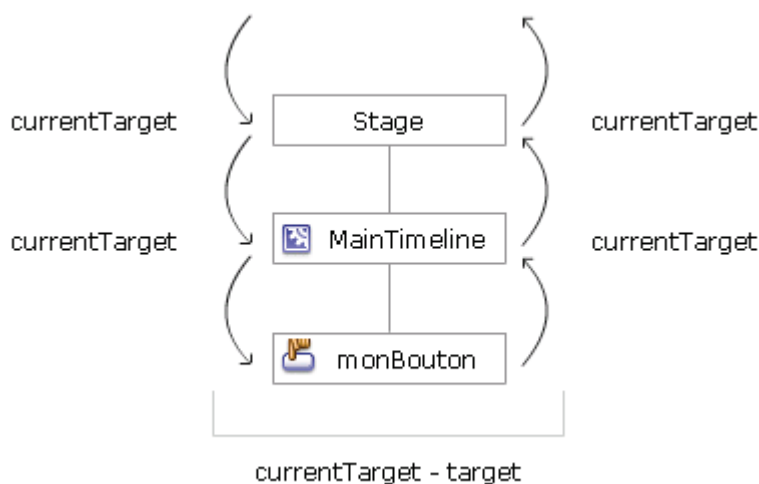
---

La propriété `currentTarget` renvoie *toujours* l'objet sur lequel nous avons appelé la méthode `addEventListener`.

---

Grace aux propriétés `target` et `currentTarget` nous pouvons donc savoir vers quel objet graphique se dirige l'événement ainsi que le nœud traité actuellement.

La figure 6-3 illustre l'idée :



*Figure 6-3. Propagation de l'événement au sein des objets nœuds.*

En testant le code suivant, nous voyons que le scénario principal est notifié de l'événement `MouseEvent.CLICK` se dirigeant vers l'objet cible :

```
// souscription à l'événement MouseEvent.CLICK auprès  
// du scénario principal pour la phase de capture
```

```
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
{
    /*
    // affiche :
    Objet cible : [object SimpleButton]
    Noeud en cours : [object MainTimeline]
    */
    trace( "Objet cible : " + pEvt.target )
    trace( "Noeud en cours : " + pEvt.currentTarget );
}
```

Bien que nous connaissions les objets graphiques associés à la propagation de l'événement, nous ne pouvons pas déterminer pour le moment la phase en cours.

L'événement actuellement diffusé correspond t'il à la phase de capture, cible, ou bien de remontée ?

### Déterminer la phase en cours

Afin de savoir à quelle phase correspond un événement nous utilisons la propriété `eventPhase` de l'objet événementiel.

Durant la phase de capture, la propriété `eventPhase` vaut 1, 2 pour la phase cible et 3 pour la phase de remontée.

Dans le même document que précédemment nous écoutons l'événement `MouseEvent.CLICK` auprès du scénario principal durant la phase de capture :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
{
    // affiche : Phase en cours : 1
    trace ( "Phase en cours : " + pEvt.eventPhase );
}
```

Lorsque nous cliquons sur notre bouton, la propriété `eventPhase` de l'objet événementiel vaut 1.

Pour afficher une information relative à la phase en cours nous pourrions être tentés d'écrire le code suivant :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
```

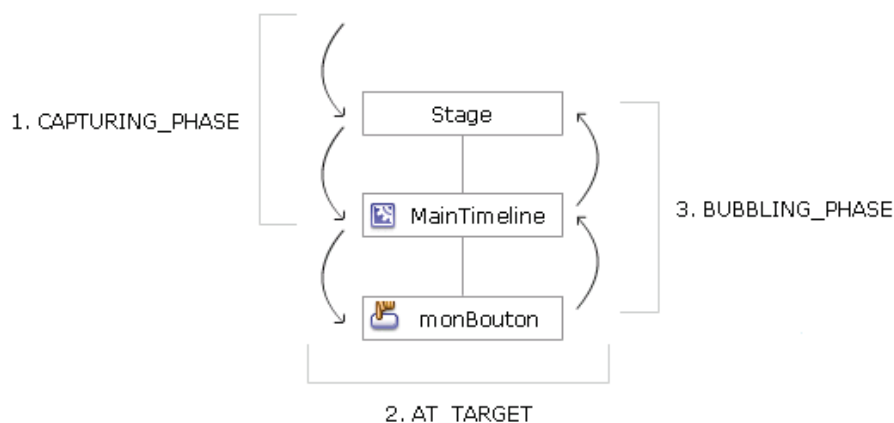
```
{  
  
    // affiche : phase de capture  
    if ( pEvt.eventPhase == 1 ) trace("phase de capture");  
  
}
```

Attention, comme pour la souscription d'événements, nous utilisons toujours des constantes de classes pour déterminer la phase en cours.

Pour cela, la classe `flash.events.EventPhase` contient trois propriétés associées à chacune des phases. En testant la valeur de chaque propriété nous récupérerons les valeurs correspondantes :

```
// affiche : 1  
trace( EventPhase.CAPTURING_PHASE );  
  
// affiche : 2  
trace( EventPhase.AT_TARGET );  
  
// affiche : 3  
trace( EventPhase.BUBBLING_PHASE );
```

La figure 6-4 illustre les constantes de la classe `EventPhase` liées à chaque phase :



*Figure 6-4. Les trois phases de la propagation événementielle.*

En prenant cela en considération, nous pouvons réécrire la fonction écouteur `clicBouton` de la manière suivante :

```
function clicBouton ( pEvt:MouseEvent )  
{  
  
    switch ( pEvt.eventPhase )  
    {  
  
        case EventPhase.CAPTURING_PHASE :  
  
    }
```

```
        trace("phase de capture");  
        break;  
  
        case EventPhase.AT_TARGET :  
            trace("phase cible");  
            break;  
  
        case EventPhase.BUBBLING_PHASE :  
            trace("phase de remontée");  
            break;  
    }  
}
```

Tout cela reste relativement abstrait et nous pouvons nous demander l'intérêt d'un tel mécanisme dans le développement d'applications `ActionScript 3`.

Grâce à ce processus nous allons rendre notre code souple et optimisé. Nous allons en tirer profit au sein d'une application dans la partie suivante.

## A retenir

- La phase de capture démarre de l'objet `Stage` jusqu'au parent de l'objet cible.
- La phase de capture ne concerne que les objets parents.
- La propriété `target` de l'objet événementiel renvoie toujours une référence vers l'objet cible.
- La propriété `currentTarget` de l'objet événementiel renvoie toujours une référence vers l'objet sur lequel nous avons appelé la méthode `addEventListener`.
- Durant la propagation d'un événement, la propriété `target` ne change pas et fait toujours référence au même objet graphique (objet cible).
- La propriété `eventPhase` de l'objet événementiel renvoie une valeur allant de 1 à 3 associées à chaque phase : `CAPTURING_PHASE`, `AT_TARGET` et `BUBBLING_PHASE`.
- Pour déterminer la phase liée à un événement nous comparons la propriété `eventPhase` aux trois constantes de la classe `flash.events.EventPhase`.

## Optimiser le code avec la phase de capture

Afin de mettre en évidence l'intérêt de la phase de capture, nous allons prendre un cas simple d'application `ActionScript` où nous souhaitons écouter l'événement `MouseEvent.CLICK` auprès de différents

boutons. Lorsque nous cliquons sur chacun d'entre eux, nous les supprimons de la liste d'affichage.

Dans un nouveau document Flash CS3 nous créons un symbole bouton de forme rectangulaire lié à une classe `Fenetre` par le panneau *Propriétés de liaison*.

Puis nous ajoutons plusieurs instances de celle-ci à notre scénario principal :

```
// nombre de fenêtres
var lng:int = 12;

var maFenetre:Fenetre;

for ( var i:int = 0; i< lng; i++ )
{

    maFenetre = new Fenetre();

    maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    addChild ( maFenetre );

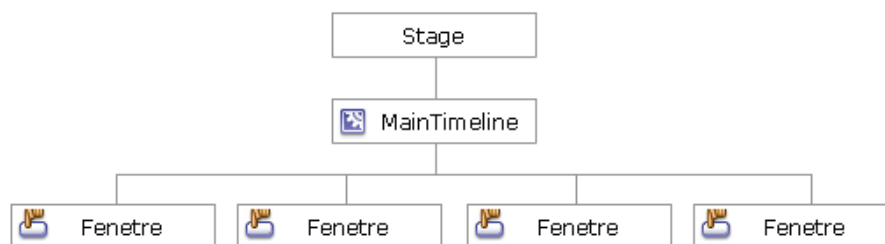
}
```

Nous obtenons le résultat illustré par la figure 6-5 :



*Figure 6-5. Instances de la classe `Fenetre`.*

Notre liste d’affichage pourrait être représentée de la manière suivante :



*Figure 6-6. Liste d’affichage avec instances de la classe Fenetre.*

Chaque clic sur une instance de symbole `Fenetre` provoque une propagation de l’événement `MouseEvent.CLICK` de l’objet `Stage` jusqu’à notre scénario principal `MainTimeline`.

Ce dernier est donc notifié de chaque clic sur nos boutons durant la phase de capture.

Comme nous l’avons vu précédemment, la phase de capture présente l’intérêt de pouvoir intercepter un événement durant sa phase descendante auprès d’un objet graphique parent.

Cela nous permet donc de centraliser l’écoute de l’événement `MouseEvent.CLICK` en appelant une seule fois la méthode `addEventListener`.

De la même manière, pour arrêter l’écoute d’un événement, nous appelons la méthode `removeEventListener` sur l’objet graphique parent seulement.

Ainsi nous n’avons pas besoin de souscrire un écouteur auprès de chaque instance de la classe `Fenetre` mais seulement auprès du scénario principal :

```

// nombre de fenêtres
var lng:int = 12;

var maFenetre:Fenetre;

for ( var i:int = 0; i< lng; i++ )
{

    maFenetre = new Fenetre();

    maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    addChild ( maFenetre );
  }

```

```
}  
  
// souscription à l'événement MouseEvent.CLICK auprès  
// du scénario principal pour la phase de capture  
addEventListener ( MouseEvent.CLICK, clicFenetre, true );  
  
function clicFenetre ( pEvt:MouseEvent ):void  
{  
  
    // affiche : [MouseEvent type="click" bubbles=true cancelable=false  
    eventPhase=1 localX=85 localY=15 stageX=92 stageY=118 relatedObject=null  
    ctrlKey=false altKey=false shiftKey=false delta=0]  
    trace( pEvt );  
  
}
```

Si nous n'avions pas utilisé la phase de capture, nous aurions du souscrire un écouteur auprès de chaque symbole `Fenetre`.

En ajoutant un écouteur auprès de l'objet parent nous pouvons capturer l'événement provenant des objets enfants, rendant ainsi notre code centralisé. Si les boutons viennent à être supprimés nous n'avons pas besoin d'appeler la méthode `removeEventListener` sur chacun d'entre eux afin de libérer les ressources, car seul le parent est écouté.

Si par la suite les boutons sont recréés, aucun code supplémentaire n'est nécessaire.

Attention, en capturant l'événement auprès du scénario principal nous écoutons tous les événements `MouseEvent.CLICK` des objets interactifs enfants. Si nous souhaitons seulement écouter les événements provenant des instances de la classe `Fenetre`, il nous faut alors les isoler dans un objet conteneur, et procéder à la capture des événements auprès de ce dernier.

Nous préférons donc l'approche suivante :

```
// nombre de fenêtres  
var lng:int = 12;  
  
// création d'un conteneur  
var conteneurFenetres:Sprite = new Sprite();  
  
// ajout à la liste d'affichage  
addChild ( conteneurFenetres );  
  
var maFenetre:Fenetre;  
  
for ( var i:int = 0; i< lng; i++ )  
{  
  
    maFenetre = new Fenetre();  
  
    maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
```

```
maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

conteneurFenetres.addChild ( maFenetre );

}

// souscription à l'événement MouseEvent.CLICK auprès
// du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, clicFenetre, true );

function clicFenetre ( pEvt:MouseEvent ):void

{

    // affiche : [MouseEvent type="click" bubbles=true cancelable=false
    eventPhase=1 localX=119 localY=46 stageX=126 stageY=53 relatedObject=null
    ctrlKey=false altKey=false shiftKey=false delta=0]
    trace( pEvt );

}
```

Afin de supprimer de l’affichage chaque fenêtre cliquée, nous ajoutons l’instruction `removeChild` au sein de la fonction écouteur `clicFenetre` :

```
function clicFenetre ( pEvt:MouseEvent ):void

{

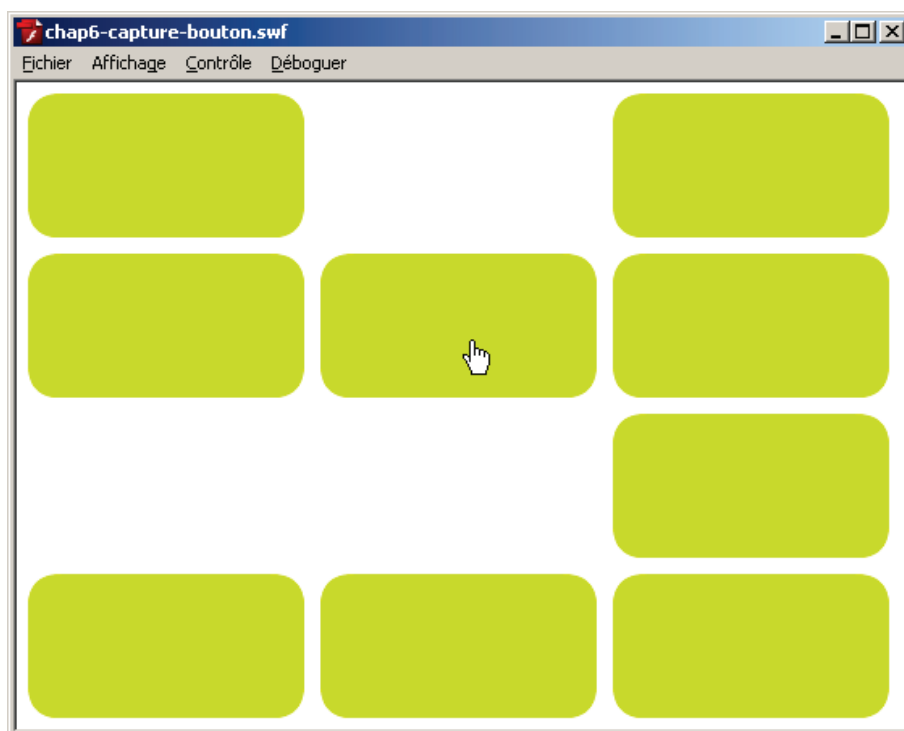
    // l'instance de Fenetre cliquée est supprimée de l'affichage
    pEvt.currentTarget.removeChild ( DisplayObject ( pEvt.target ) );

}
```

La propriété `target` fait ainsi référence à l’instance de `Fenetre` cliquée, tandis que la propriété `currentTarget` référence le conteneur.

A chaque clic, l’instance cliquée est supprimée de la liste d’affichage :





*Figure 6-7. Suppression d'instances de la classe  
Fenetre.*

Nous reviendrons tout au long de l'ouvrage sur l'utilisation de la phase de capture afin de maîtriser totalement ce concept.

Nous allons nous attarder à présent sur la phase cible.

### A retenir

- Grâce à la phase de capture, nous souscrivons l'écouteur auprès de l'objet graphique parent afin de capturer les événements des objets enfants.
- Notre code est plus optimisé et plus centralisé.

## La phase cible

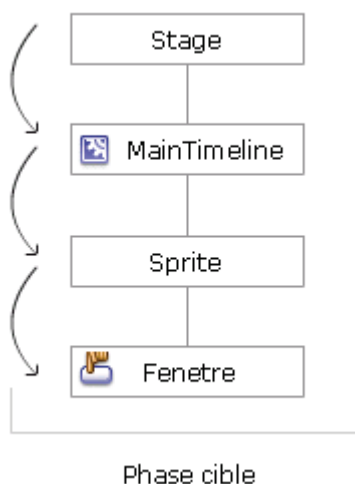
La phase cible correspond à la diffusion de l'événement depuis *l'objet cible*. Certains événements associés à des objets graphiques ne participent qu'à celle-ci.

C'est le cas des quatre événements suivants qui ne participent ni à la phase de capture ni à la phase de remontée :

- `Event.ENTER_FRAME`
- `Event.ACTIVATE`

- `Event.DEACTIVATE`
- `Event.RENDER`

Nous allons reprendre l'exemple précédent, en préférant cette fois la phase cible à la phase de capture :



*Figure 6-8. Phase cible.*

Afin d'écouter l'événement `MouseEvent.CLICK`, nous souscrivons un écouteur auprès de chaque instance de la classe `Fenetre` :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurFenetres );

var maFenetre:Fenetre;

for (var i:int = 0; i< lng; i++ )
{

    maFenetre = new Fenetre();

    // souscription auprès de chaque instance pour la phase cible
    maFenetre.addEventListener ( MouseEvent.CLICK, clicFenetre );

    maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    conteneurFenetres.addChild ( maFenetre );

}

function clicFenetre ( pEvt:MouseEvent ):void
```

```
{
  // affiche : [object Fenetre]
  trace( pEvt.target );
}
```

Au clic, l'événement commence sa propagation puis atteint l'objet cible, notre fonction écouteur `clicFenetre` est déclenchée.

---

Ainsi, lorsque nous appelons la méthode `addEventListener` sans spécifier le paramètre `useCapture`, nous souscrivons l'écouteur à la phase cible ainsi qu'à la phase de remontée que nous traiterons juste après.

---

Si nous affichons le contenu de la propriété `target` de l'objet événementiel, celui-ci nous renvoie une référence vers l'objet cible, ici notre objet `Fenetre`.

Pour supprimer chaque instance, nous rajoutons l'instruction `removeChild` au sein de notre fonction écouteur en passant l'objet référencé par la propriété `target` de l'objet événementiel :

```
function clicFenetre ( pEvt:MouseEvent ):void
{
  // l'instance de Fenetre cliquée est supprimée de l'affichage
  conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

  // désinscription de la fonction clicFenetre à l'événement
  // MouseEvent.CLICK
  pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );
}
```

Il est important de noter que durant la phase cible, l'objet cible est aussi le diffuseur de l'événement.

En testant le code suivant, nous voyons que les propriétés `target` et `currentTarget` de l'objet événementiel référencent le même objet graphique :

```
function clicFenetre ( pEvt:MouseEvent ):void
{
  // affiche : true
  trace( pEvt.target == pEvt.currentTarget );

  // l'instance de Fenetre cliquée est supprimée de l'affichage
  conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

  // désinscription de la fonction clicFenetre à l'événement
  // MouseEvent.CLICK
}
```

```
pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );  
}
```

Pour tester si l'événement diffusé est issu de la phase cible nous pouvons comparer la propriété `eventPhase` de l'objet événementiel à la constante `EventPhase.AT_TARGET` :

```
function clicFenetre ( pEvt:MouseEvent ):void  
{  
    // affiche : true  
    trace( pEvt.target == pEvt.currentTarget );  
  
    // affiche : phase cible  
    if ( pEvt.eventPhase == EventPhase.AT_TARGET )  
    {  
        trace("phase cible");  
    }  
  
    // l'instance de Fenetre cliquée est supprimée de l'affichage  
    conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );  
  
    // désinscription de la fonction clicFenetre à l'événement  
    // MouseEvent.CLICK  
    pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );  
}
```

En utilisant la phase cible nous avons perdu en souplesse en souscrivant la fonction écouteur `clicFenetre` auprès de chaque instance de la classe `Fenetre`, et géré la désinscription des écouteurs lors de la suppression de la liste d'affichage.

Bien entendu, la phase de capture ne doit et ne peut pas être utilisée systématiquement. En utilisant la phase cible nous pouvons enregistrer une fonction écouteur à un bouton unique, ce qui peut s'avérer primordial lorsque la logique associée à chaque bouton est radicalement différente. Bien que très pratique dans certains cas, la phase de capture intercepte les clics de chaque bouton en exécutant une seule fonction écouteur ce qui peut s'avérer limitatif dans certaines situations.

Chacune des phases doit donc être considérée et utilisée lorsque la situation vous paraît la plus appropriée.

---

## A retenir

---

- La phase cible correspond à la diffusion de l'événement par l'objet cible.
- Les objets graphiques diffusent des événements pouvant se propager.
- Les objets non graphiques diffusent des événements ne participant qu'à la phase cible.
- Il convient de bien étudier l'imbrication des objets graphiques au sein de l'application, et d'utiliser la phase la plus adaptée à nos besoins.

ActionScript 3 offre en plus la possibilité d'intervenir sur la propagation d'un événement. Cette fonctionnalité peut être utile lorsque nous souhaitons empêcher un événement d'atteindre l'objet cible en empêchant la poursuite de sa propagation.

Nous allons découvrir à l'aide d'un exemple concret comment utiliser cette nouvelle notion.

## Intervenir sur la propagation

Il est possible de stopper la propagation d'un événement depuis n'importe quel nœud. Quel pourrait être l'intérêt de stopper la propagation d'un événement durant sa phase de capture ou sa phase de remontée ?

Dans certains cas nous avons besoin de verrouiller l'ensemble d'une application, comme par exemple la sélection d'éléments interactifs dans une application ou dans un jeu.

Deux méthodes définies par la classe `flash.events.Event` permettent d'intervenir sur la propagation :

- `objtEvenementiel.stopPropagation()`
- `objtEvenementiel.stopImmediatePropagation()`

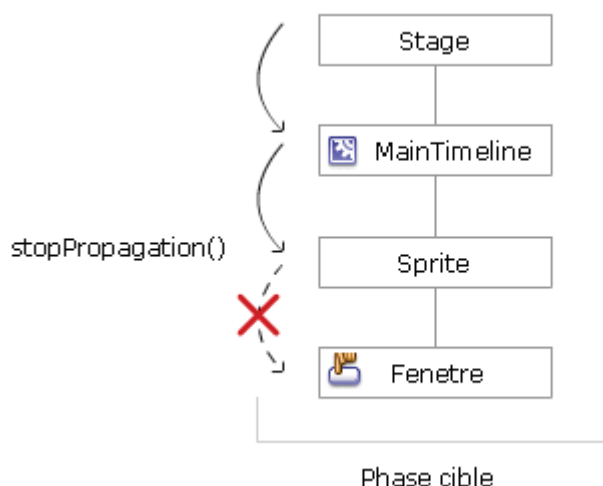
Ces deux méthodes sont quasiment identiques mais diffèrent quelque peu, nous allons nous y attarder à présent.

Imaginons que nous souhaitions verrouiller tous les boutons de notre exemple précédent.

Dans l'exemple suivant nous allons intervenir sur la propagation d'un événement `MouseEvent.CLICK` durant la phase de capture. En stoppant sa propagation au niveau du conteneur, nous allons l'empêcher d'atteindre l'objet cible.

Ainsi la phase cible ne sera jamais atteinte.

La figure 6-9 illustre l'interruption de propagation d'un événement :



*Figure 6-9. Intervention sur la propagation d'un événement.*

Afin d'intercepter l'événement `MouseEvent.CLICK` avant qu'il ne parvienne jusqu'à l'objet cible, nous écoutons l'événement `MouseEvent.CLICK` lors de la phase de capture auprès du conteneur `conteneurFenetres` :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurFenetres );

var maFenetre:Fenetre;

for (var i:int = 0; i< lng; i++ )
{

    maFenetre = new Fenetre();

    // souscription auprès de chaque instance pour la phase cible
    maFenetre.addEventListener ( MouseEvent.CLICK, clicFenetre );

    maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    conteneurFenetres.addChild ( maFenetre );

}

// souscription à l'événement MouseEvent.CLICK auprès
// du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );
```

```
function clicFenetre ( pEvt:Event ):void
{
    // l'instance de Fenetre cliquée est supprimée de l'affichage
    conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

    // désinscription de la fonction clicFenetre à l'événement
    // MouseEvent.CLICK
    pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );
}

function captureClic ( pEvt:MouseEvent ):void
{
    // affiche : Capture de l'événement : click
    trace("Capture de l'événement : " + pEvt.type );
}
```

Nous remarquons que la fonction `captureClic` est bien déclenchée à chaque clic bouton.

Pour stopper la propagation, nous appelons la méthode `stopPropagation` sur l'objet événementiel diffusé :

```
function captureClic ( pEvt:MouseEvent ):void
{
    // affiche : Objet cible : [object Fenetre]
    trace( "Objet cible : " + pEvt.target );

    // affiche : Objet notifié : [object Sprite]
    trace( "Objet notifié : " + pEvt.currentTarget );

    // affiche : Capture de l'événement : click
    trace("Capture de l'événement : " + pEvt.type );

    pEvt.stopPropagation();
}
```

Lorsque la méthode `stopPropagation` est exécutée, l'événement interrompt sa propagation sur le nœud en cours, la fonction `clicFenetre` n'est plus déclenchée.

Grâce au code suivant, les instances de la classe `Fenetre` sont supprimées uniquement lorsque la touche ALT est enfoncée :

```
function captureClic ( pEvt:MouseEvent ):void
{
    // affiche : Objet cible : [object Fenetre]
    trace( "Objet cible : " + pEvt.target );
```

```
// affiche : Objet notifié : [object Sprite]
trace( "Objet notifié : " + pEvt.currentTarget );

// affiche : Capture de l'événement : click
trace("Capture de l'événement : " + pEvt.type );

if ( !pEvt.altKey ) pEvt.stopPropagation();

}
```

Nous reviendrons sur les propriétés de la classe `MouseEvent` au cours du prochain chapitre intitulé *Interactivité*.

Attention, la méthode `stopPropagation` interrompt la propagation de l'événement auprès des nœuds suivants, mais autorise l'exécution des écouteurs souscrits au nœud en cours. Si nous ajoutons un autre écouteur à l'objet `conteneurFenetres`, bien que la propagation soit stoppée, la fonction écouteur est déclenchée.

Afin de mettre en évidence la méthode `stopImmediatePropagation` nous ajoutons une fonction `captureClicBis` comme autre écouteur du conteneur :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurFenetres );

var maFenetre:Fenetre;

for (var i:int = 0; i< lng; i++ )

{

    maFenetre = new Fenetre();

    // souscription auprès de chaque instance pour la phase cible
    maFenetre.addEventListener ( MouseEvent.CLICK, clicFenetre );

    maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    conteneurFenetres.addChild ( maFenetre );

}

// souscription à l'événement MouseEvent.CLICK auprès
// du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );

// souscription à l'événement MouseEvent.CLICK auprès
// du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClicBis, true );
};
```



```
function clicFenetre ( pEvt:Event ):void
{
    // l'instance de Fenetre cliquée est supprimée de l'affichage
    conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

    // désinscription de la fonction clicFenetre à l'événement
    // MouseEvent.CLICK
    pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );
}

function captureClic ( pEvt:MouseEvent ):void
{
    // affiche : Objet cible : [object Fenetre]
    trace( "Objet cible : " + pEvt.target );
    // affiche : Objet notifié : [object Sprite]
    trace( "Objet notifié : " + pEvt.currentTarget );
    // affiche : Capture de l'événement : click
    trace("Capture de l'événement : " + pEvt.type );

    if ( !pEvt.altKey ) pEvt.stopPropagation();
}

function captureClicBis ( pEvt:MouseEvent ):void
{
    trace("fonction écouteur captureClicBis déclenchée");
}
```

En testant le code précédent, nous voyons que notre fonction `captureClicBis` est bien déclenchée bien que la propagation de l'événement soit interrompue. A l'inverse, si nous appelons la méthode `stopImmediatePropagation`, la fonction `captureClicBis` n'est plus déclenchée :

```
function captureClic ( pEvt:MouseEvent ):void
{
    // affiche : Objet cible : [object Fenetre]
    trace( "Objet cible : " + pEvt.target );

    // affiche : Objet notifié : [object Sprite]
    trace( "Objet notifié : " + pEvt.currentTarget );

    // affiche : Capture de l'événement : click
    trace("Capture de l'événement : " + pEvt.type );

    if ( !pEvt.altKey ) pEvt.stopImmediatePropagation();
}
```

Grâce aux méthodes `stopPropagation` et `stopImmediatePropagation` nous pouvons ainsi maîtriser le flot événementiel avec précision.

## A retenir

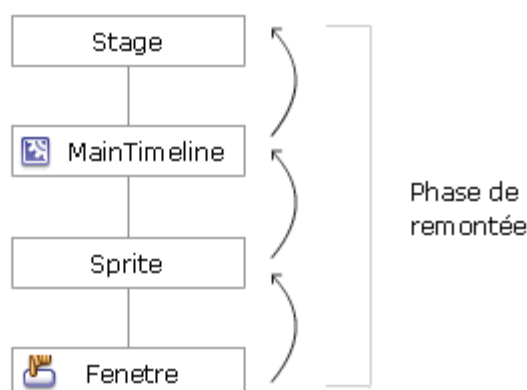
- Il est possible d'intervenir sur la propagation d'un événement à l'aide des méthodes `stopPropagation` et `stopImmediatePropagation`.
- La méthode `stopPropagation` interrompt la propagation d'un événement mais n'empêche pas sa diffusion auprès des écouteurs du nœud en cours.
- La méthode `stopImmediatePropagation` interrompt la propagation d'un événement et empêche sa diffusion même auprès des écouteurs du même nœud.

## La phase de remontée

La phase de remontée constitue la dernière phase de la propagation. Durant cette phase, l'événement parcourt le chemin inverse de la phase de capture, notifiant chaque nœud parent de l'objet cible.

Le parcours de l'événement durant cette phase s'apparente à celui d'une bulle remontant à la surface de l'eau. C'est pour cette raison que cette phase est appelée en anglais « *bubbling phase* ». L'événement remontant de l'objet cible jusqu'à l'objet `Stage`.

La figure 6-10 illustre la phase de remontée :



*Figure 6-10. Phase de remontée.*

Pour souscrire un écouteur auprès de la phase de remontée nous appelons la méthode `addEventListener` sur un des objets

graphiques parent en passant la valeur booléenne `false` au paramètre `useCapture` :

```
| monObjetParent.addEventListener ( MouseEvent.CLICK, clicRemontee, false );
```

En lisant la signature de la méthode `addEventListener` nous voyons que le paramètre `useCapture` vaut `false` par défaut :

```
| public function addEventListener(type:String, listener:Function,  
| useCapture:Boolean = false, priority:int = 0, useWeakReference:Boolean =  
| false):void
```

Ainsi, la souscription auprès de la phase de remontée peut s'écrire sans préciser le paramètre `useCapture` :

```
| objetParent.addEventListener ( MouseEvent.CLICK, clicRemontee );
```

Cela signifie que lorsque nous écoutons un événement pour la phase cible, la phase de remontée est automatiquement écoutée.

La fonction `clicRemontee` recevra donc aussi les événements `MouseEvent.CLICK` provenant des enfants durant leur phase de remontée.

Dans l'exemple suivant, nous écoutons l'événement `MouseEvent.CLICK` lors de la phase de capture et de remontée auprès du conteneur :

```
// nombre de fenêtres  
var lng:int = 12;  
  
// création d'un conteneur  
var conteneurFenetres:Sprite = new Sprite();  
  
// ajout à la liste d'affichage  
addChild ( conteneurFenetres );  
  
var maFenetre:Fenetre;  
  
for (var i:int = 0; i< lng; i++ )  
{  
  
    maFenetre = new Fenetre();  
  
    maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );  
    maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );  
  
    conteneurFenetres.addChild ( maFenetre );  
  
}  
  
// souscription auprès du conteneur pour la phase de capture  
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );  
  
// souscription auprès du conteneur pour la phase de remontée  
conteneurFenetres.addEventListener ( MouseEvent.CLICK, clicRemontee );
```

```
function captureClic ( pEvt:MouseEvent ):void
{
    /* affiche :
    phase de capture de l'événement : click
    noeud en cours de notification : [object Sprite]
    */
    if ( pEvt.eventPhase == EventPhase.CAPTURING_PHASE )
    {
        trace("phase de capture de l'événement : " + pEvt.type );
        trace("noeud en cours de notification : " + pEvt.currentTarget );
    }
}

function clicRemontee ( pEvt:MouseEvent ):void
{
    /* affiche :
    phase de remontée de l'événement : click
    noeud en cours de notification : [object Sprite]
    */
    if ( pEvt.eventPhase == EventPhase.BUBBLING_PHASE )
    {
        trace("phase de remontée de l'événement : " + pEvt.type );
        trace("noeud en cours de notification : " + pEvt.currentTarget );
    }
}
```

Nous allons aller plus loin en ajoutant une réorganisation automatique des fenêtres lorsqu'une d'entre elles est supprimée.

Pour cela nous modifions le code en intégrant un effet d'inertie afin de disposer chaque instance de la classe `Fenetre` avec un effet de ralenti :

```
// modification de la cadence de l'animation
stage.frameRate = 30;

for (var i:int = 0; i < lng; i++ )
{
    maFenetre = new Fenetre();

    maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

    conteneurFenetres.addChild ( maFenetre );
}
```

```

    }

    function mouvement ( pEvt:Event ):void
    {
        // algorithme d'inertie
        pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
        pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;
    }

```

Puis nous intégrons la logique nécessaire au sein des fonctions écouteurs `captureClic` et `clicRemontee` :

```

// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurFenetres );

var maFenetre:Fenetre;

// modification de la cadence de l'animation
stage.frameRate = 30;

for (var i:int = 0; i < lng; i++ )
{
    maFenetre = new Fenetre();

    maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

    conteneurFenetres.addChild ( maFenetre );
}

function mouvement ( pEvt:Event ):void
{
    // algorithme d'inertie
    pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
    pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;
}

// souscription auprès du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );

// souscription auprès du conteneur pour la phase de remontée
conteneurFenetres.addEventListener ( MouseEvent.CLICK, clicRemontee );

```

```
function captureClic ( pEvt:MouseEvent ):void
{
    pEvt.currentTarget.removeChild ( DisplayObject ( pEvt.target ) );
}

function clicRemontee ( pEvt:MouseEvent ):void
{
    var lng:int = pEvt.currentTarget.numChildren;

    var objetGraphique:DisplayObject;
    var maFenetre:Fenetre;

    while ( lng-- )
    {
        // récupération des objets graphiques
        objetGraphique = pEvt.currentTarget.getChildAt ( lng );

        // si l'un d'entre eux est de type Fenetre
        if ( objetGraphique is Fenetre )
        {
            // nous le transtypons en type Fenetre
            maFenetre = Fenetre ( objetGraphique );

            // repositionnement de chaque occurrence
            maFenetre.destX = 7 + Math.round ( lng % 3 ) * ( maFenetre.width
+ 10 );
            maFenetre.destY = 7 + Math.floor ( lng / 3 ) * ( maFenetre.height
+ 10 );
        }
    }
}
```

Nous profitons de la phase de remontée pour que le conteneur réorganise ses objets enfants lorsque l'un d'entre eux a été supprimé de la liste d'affichage.

Nous verrons au cours du chapitre 8 intitulé *Programmation orientée objet* comment notre code actuel pourrait être encore amélioré.

## Ecouter plusieurs phases

Afin de suivre la propagation d'un événement, nous pouvons souscrire un écouteur auprès de chacune des phases. En affichant les valeurs des propriétés `eventPhase`, `target` et `currentTarget` de l'objet événementiel nous obtenons le chemin parcouru par l'événement.

Dans le code suivant, la fonction `ecoutePhase` est souscrite auprès des trois phases de l'événement `MouseEvent.CLICK` :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurFenetres );

var maFenetre:Fenetre;

// modification de la cadence de l'animation
stage.frameRate = 30;

for (var i:int = 0; i< lng; i++ )
{
    maFenetre = new Fenetre();

    maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    // souscription auprès de chaque instance pour la phase cible
    maFenetre.addEventListener ( MouseEvent.CLICK, ecoutePhase );

    maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

    conteneurFenetres.addChild ( maFenetre );
}

function mouvement ( pEvt:Event ):void
{
    // algorithme d'inertie
    pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
    pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;
}

// souscription auprès du conteneur pour la phase de remontée
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase );

// souscription auprès du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase, true );

function ecoutePhase ( pEvt:MouseEvent ):void
{
    /* affiche :
    1 : Phase de capture de l'événement : click
    Objet cible : [object Fenetre]
    Objet notifié : [object Sprite]
    2 : Phase cible de l'événement : click
```

```
Objet cible : [object Fenetre]
Objet notifié : [object Fenetre]
3 : Phase de remontée de l'événement : click
Objet cible : [object Fenetre]
Objet notifié : [object Sprite]
*/

switch ( pEvt.eventPhase )

{

    case EventPhase.CAPTURING_PHASE :
        trace ( pEvt.eventPhase + " : Phase de capture de l'événement : "
+ pEvt.type );
        break;

    case EventPhase.AT_TARGET :
        trace ( pEvt.eventPhase + " : Phase cible de l'événement : " +
pEvt.type );
        break;

    case EventPhase.BUBBLING_PHASE :
        trace ( pEvt.eventPhase + " : Phase de remontée de l'événement :
" + pEvt.type );
        break;

}

trace( "Objet cible : " + pEvt.target );
trace( "Objet notifié : " + pEvt.currentTarget );

}
```

Le panneau de sortie nous affiche l'historique de la propagation de l'événement `MouseEvent.CLICK`.

## A retenir

- A l'instar de la phase de capture, la phase de remontée ne concerne que les objets parents.
- La phase de remontée permet à un objet parent d'être notifié d'un événement provenant de l'un de ses enfants.
- Un même écouteur peut être souscrit aux différentes phases d'un événement.

## La propriété `Event.bubbles`

Pour savoir si un événement participe ou non à la phase de remontée nous pouvons utiliser la propriété `bubbles` de l'objet événementiel.

Nous pouvons mettre à jour la fonction `ecoutePhase` afin de savoir si l'événement en cours participe à la phase de remontée :

```
function ecoutePhase ( pEvt:MouseEvent ):void

{
```



```
/* affiche :
1 : Phase de capture de l'événement : click
Objet cible : [object Fenetre]
Objet notifié : [object Sprite]
Événement participant à la phase de remontée : true
2 : Phase cible de l'événement : click
Objet cible : [object Fenetre]
Objet notifié : [object Fenetre]
Événement participant à la phase de remontée : true
3 : Phase de remontée de l'événement : click
Objet cible : [object Fenetre]
Objet notifié : [object Sprite]
Événement participant à la phase de remontée : true
*/

switch ( pEvt.eventPhase )
{
    case EventPhase.CAPTURING_PHASE :
        trace ( pEvt.eventPhase + " : Phase de capture de l'événement : " +
pEvt.type );
        break;

    case EventPhase.AT_TARGET :
        trace ( pEvt.eventPhase + " : Phase cible de l'événement : " +
pEvt.type );
        break;

    case EventPhase.BUBBLING_PHASE :
        trace ( pEvt.eventPhase + " : Phase de remontée de l'événement :
" + pEvt.type );
        break;

}

trace( "Objet cible : " + pEvt.target );
trace( "Objet notifié : " + pEvt.currentTarget );

trace ( "Événement participant à la phase de remontée : " + pEvt.bubbles );
}
```

Notons que tout événement se propageant vers le haut participe forcément aux phases de capture et cible.

## Suppression d'écouteurs

Comme nous l'avons traité lors du chapitre 3 intitulé *Le modèle événementiel*, lorsque nous souhaitons arrêter l'écoute d'un événement nous utilisons la méthode `removeEventListener`.

Lorsque nous avons souscrit un écouteur pour une phase spécifique nous devons spécifier la même phase lors de l'appel de la méthode `removeEventListener`.

Si nous écoutons un événement pour la phase de capture :

```
objetParent.addEventListener ( MouseEvent.CLICK, clicBouton, true );
```

Pour supprimer l'écoute nous devons spécifier la phase concernée :

```
objetParent.removeEventListener ( MouseEvent.CLICK, clicBouton, true );
```

Il en est de même pour les phases de remontée et cible :

```
objetParent.removeEventListener ( MouseEvent.CLICK, clicBouton, false );
```

```
objetCible.removeEventListener ( MouseEvent.CLICK, clicBouton, false );
```

Ou bien de manière implicite :

```
objetParent.removeEventListener ( MouseEvent.CLICK, clicBouton );
```

```
objetCible.removeEventListener ( MouseEvent.CLICK, clicBouton );
```

En intégrant cela dans notre exemple précédent, nous obtenons le code suivant :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurFenetres );

var maFenetre:Fenetre;

// modification de la cadence de l'animation
stage.frameRate = 30;

for (var i:int = 0; i< lng; i++ )
{
    maFenetre = new Fenetre();

    maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
    maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

    // souscription auprès de chaque instance pour la phase cible
    maFenetre.addEventListener ( MouseEvent.CLICK, ecoutePhase );

    // désinscription auprès de chaque occurrence pour la phase cible
    maFenetre.removeEventListener ( MouseEvent.CLICK, ecoutePhase );

    maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

    conteneurFenetres.addChild ( maFenetre );
}

function mouvement ( pEvt:Event ):void
{
    // algorithme d'inertie
```

```

    pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
    pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;
}

// souscription auprès du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase, true );

// désinscription auprès du conteneur pour la phase de capture
conteneurFenetres.removeEventListener ( MouseEvent.CLICK, ecoutePhase, true );

// souscription auprès du conteneur pour la phase de remontée
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase );

// désinscription auprès du conteneur pour la phase de remontée
conteneurFenetres.removeEventListener ( MouseEvent.CLICK, ecoutePhase );

function ecoutePhase ( pEvt:MouseEvent ):void
{
    /* affiche :
    1 : Phase de capture de l'événement : click
    Objet cible : [object Fenetre]
    Objet notifié : [object Sprite]
    2 : Phase cible de l'événement : click
    Objet cible : [object Fenetre]
    Objet notifié : [object Fenetre]
    3 : Phase de remontée de l'événement : click
    Objet cible : [object Fenetre]
    Objet notifié : [object Sprite]
    */

    switch ( pEvt.eventPhase )
    {
        case EventPhase.CAPTURING_PHASE :
            trace ( pEvt.eventPhase + " : Phase de capture de l'événement : " +
+ pEvt.type );
            break;

        case EventPhase.AT_TARGET :
            trace ( pEvt.eventPhase + " : Phase cible de l'événement : " +
pEvt.type );
            break;

        case EventPhase.BUBBLING_PHASE :
            trace ( pEvt.eventPhase + " : Phase de remontée de l'événement :
" + pEvt.type );
            break;

    }

    trace( "Objet cible : " + pEvt.target );
    trace( "Objet notifié : " + pEvt.currentTarget );

    trace ( "Événement participant à la phase de remontée : " + pEvt.bubbles );
}

```

## A retenir

- La propriété `bubbles` d'un objet événementiel permet de savoir si l'événement en cours participe à la phase de remontée.
- Lorsque nous avons souscrit un écouteur pour une phase spécifique nous devons spécifier la même phase lors de l'appel de la méthode `removeEventListener`.

Nous avons abordé au cours des chapitres précédents un ensemble de concepts clés que nous allons pouvoir mettre à profit dès maintenant.

En route pour le nouveau chapitre intitulé *Interactivité* !

# 7

## Interactivité

<b>AU CŒUR DE FLASH.....</b>	<b>2</b>
<b>INTERACTIVITE AVEC SIMPLEBUTTON .....</b>	<b>2</b>
ENRICHIR GRAPHIQUEMENT UN SIMPLEBUTTON.....	4
<b>CREER UN MENU DYNAMIQUE .....</b>	<b>9</b>
MOUVEMENT PROGRAMMATIQUE .....	14
<b>LES PIEGES DU RAMASSE-MIETTES .....</b>	<b>19</b>
<b>AJOUT DE COMPORTEMENT BOUTON .....</b>	<b>20</b>
ZONE RÉACTIVE .....	23
<b>GESTION DU FOCUS .....</b>	<b>29</b>
<b>POUR ALLER PLUS LOIN .....</b>	<b>34</b>
<b>ESPACE DE COORDONNEES .....</b>	<b>37</b>
<b>EVENEMENT GLOBAL .....</b>	<b>40</b>
<b>MISE EN APPLICATION .....</b>	<b>41</b>
<b>MISE A JOUR DU RENDU .....</b>	<b>43</b>
<b>GESTION DU CLAVIER .....</b>	<b>45</b>
DÉTERMINER LA TOUCHE APPUYÉE.....	46
GESTION DE TOUCHES SIMULTANÉES .....	48
SIMULER LA METHODE KEY.ISDOWN.....	54
<b>SUPERPOSITION .....</b>	<b>55</b>
<b>L'ÉVÉNEMENT EVENT.RESIZE.....</b>	<b>58</b>

## Au cœur de Flash

L'interactivité est l'une des forces majeures et le cœur du lecteur Flash. En un temps réduit, nous avons toujours pu réaliser une combinaison d'objets réactifs à la souris, au clavier ou autres.

La puissance apportée par ActionScript 3 introduit quelques nuances relatives à l'interactivité, que nous allons découvrir ensemble à travers différents exercices pratiques.

## Interactivité avec SimpleButton

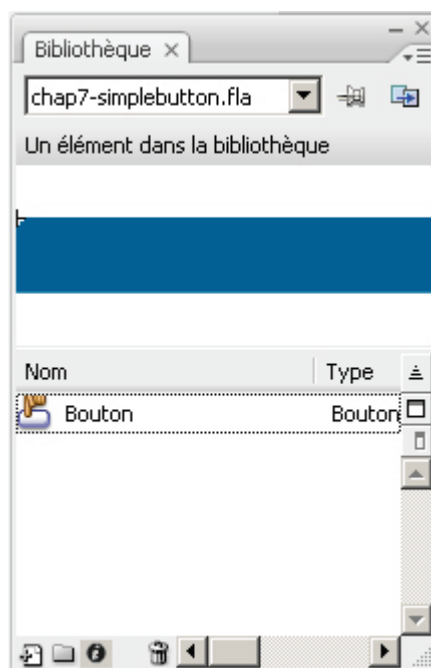
Comme nous l'avons découvert lors du chapitre 5 intitulé *Les symboles*, la classe `SimpleButton` représente les symboles de type boutons en ActionScript 3. Cette classe s'avère très pratique en offrant une gestion avancée des boutons créés depuis l'environnement auteur ou par programmation.

Il faut considérer l'objet `SimpleButton` comme un bouton constitué de quatre `DisplayObject` affectés à chacun de ses états. Chacun d'entre eux est désormais accessible par des propriétés dont voici le détail :

- `SimpleButton.upState` : définit l'état haut
- `SimpleButton.overState` : définit l'état dessus
- `SimpleButton.downState` : définit l'état abaissé
- `SimpleButton.hitTestState` : définit l'état cliqué

Pour nous familiariser avec cette nouvelle classe nous allons tout d'abord créer un simple bouton, puis à l'aide de ce dernier nous construirons un menu dynamique.

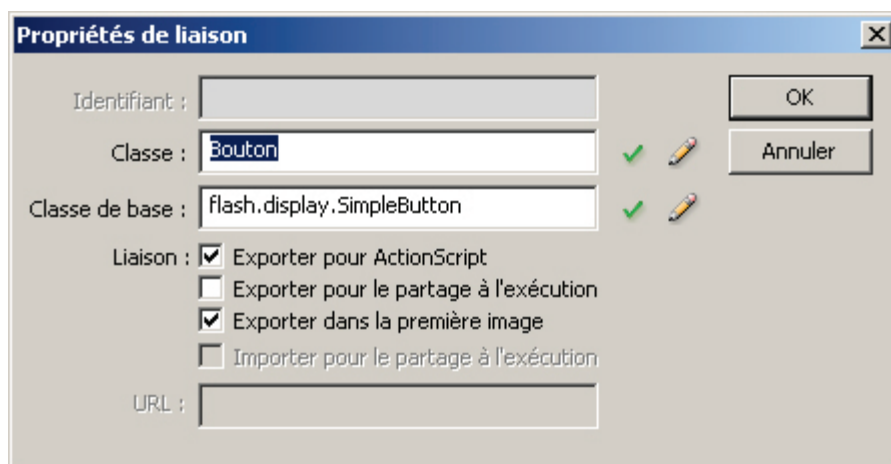
Dans un nouveau document Flash CS3, nous créons un symbole bouton, celui-ci est aussitôt ajouté à la bibliothèque :



*Figure 7-1. Symbole bouton.*

Puis nous définissons une classe `Bouton` associée, à l'aide du panneau *Propriétés de liaison*. En définissant une classe associée nous pourrions instancier plus tard notre bouton par programmation.

La figure 7-2 illustre le panneau :



*Figure 7-2. Définition de classe associée.*

Nous posons une occurrence de ce dernier sur le scénario principal et lui donnons `monBouton` comme nom d'occurrence. Chaque propriété relative à l'état du bouton renvoie un objet de type `flash.display.Shape` :

```
/*affiche :
```

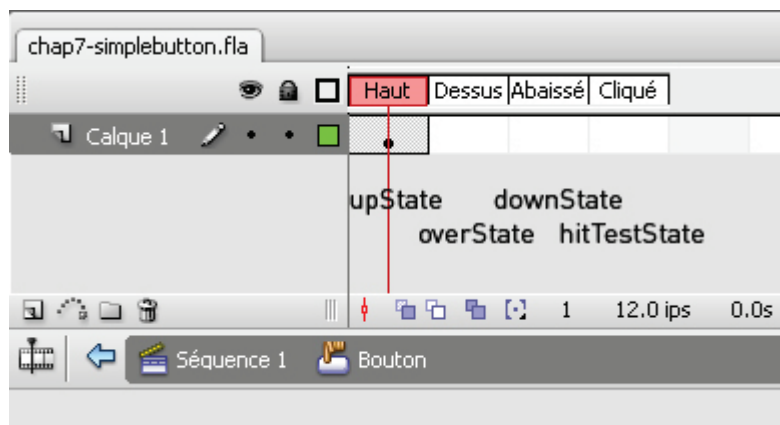
```

[object Shape]
[object Shape]
[object Shape]
[object Shape]
*/
trace( monBouton.upState );
trace( monBouton.overState );
trace( monBouton.downState );
trace( monBouton.hitTestState );

```

Chaque état peut être défini par n'importe quel objet de type `DisplayObject`. Lorsque nous créons un bouton dans l'environnement auteur et qu'une simple forme occupe l'état Haut, nous obtenons un objet `Shape` pour chaque état. Si nous avons créé un clip pour l'état Haut nous aurions récupéré un objet de type `flash.display.MovieClip`.

La figure 7-3 illustre la correspondance entre chaque état et chaque propriété :



*Figure 7-3. Correspondance des propriétés d'états.*

Il était auparavant impossible d'accéder dynamiquement aux différents états d'un bouton. Nous verrons plus tard qu'un bouton, ainsi que ces états et les sons associés, peuvent être entièrement créés ou définis par programmation. Nous reviendrons très vite sur les autres nouveautés apportées par la classe `SimpleButton`.

Pour enrichir graphiquement un bouton, nous devons comprendre comment celui-ci fonctionne, voyons à présent comment décorer notre bouton afin de le rendre utilisable pour notre menu.

## Enrichir graphiquement un SimpleButton

Un bouton ne se limite généralement pas à une simple forme cliquable, une légende ou une animation ou d'autres éléments peuvent être ajoutés afin d'enrichir graphiquement le bouton. Avant de commencer à coder, il faut savoir que la classe `SimpleButton` est



une classe particulière. Nous pourrions penser que celle-ci hérite de la classe `DisplayObjectContainer`, mais il n'en est rien.

La classe `SimpleButton` hérite de la classe `InteractiveObject` et ne peut se voir ajouter du contenu à travers les méthodes traditionnelles telles `addChild`, `addChildAt`, etc.

---

Seules les propriétés `upState`, `downState`, `overState` et `hitTestState` permettent d'ajouter et d'accéder au contenu d'une occurrence de `SimpleButton`.

---

Le seul moyen de supprimer un état du bouton est de passer la valeur `null` à un état du bouton :

```
// supprime l'état Abaissé
monBouton.downState = null;
```

Si nous testons le code précédent, nous voyons que le bouton ne dispose plus d'état abaissé lorsque nous cliquons dessus.

Pour ajouter une légende à notre occurrence de `SimpleButton` nous devons ajouter un champ texte à chaque objet graphique servant d'état. Si nous ajoutons directement un objet `TextField` à l'un des états nous le remplaçons :

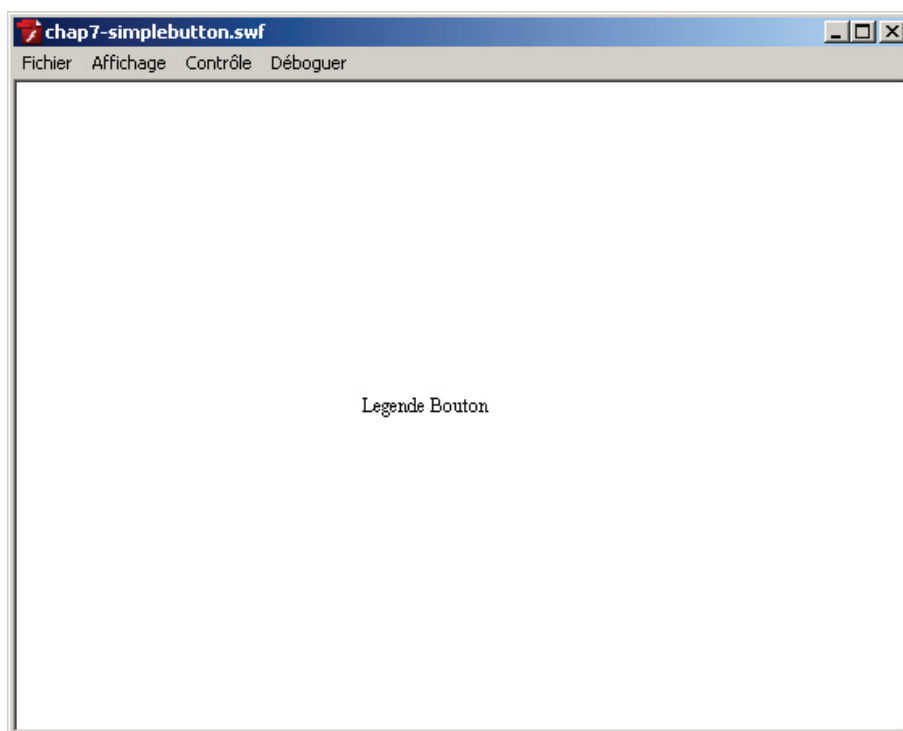
```
// création du champ texte servant de légende
var maLégende:TextField = new TextField();

// nous affectons le contenu
maLégende.text = "Légende Bouton";

// nous passons la légende en tant qu'état (Haut) du bouton
monBouton.upState = maLégende;
```

En testant le code précédent, nous obtenons un bouton cliquable constitué d'une légende comme état Haut.

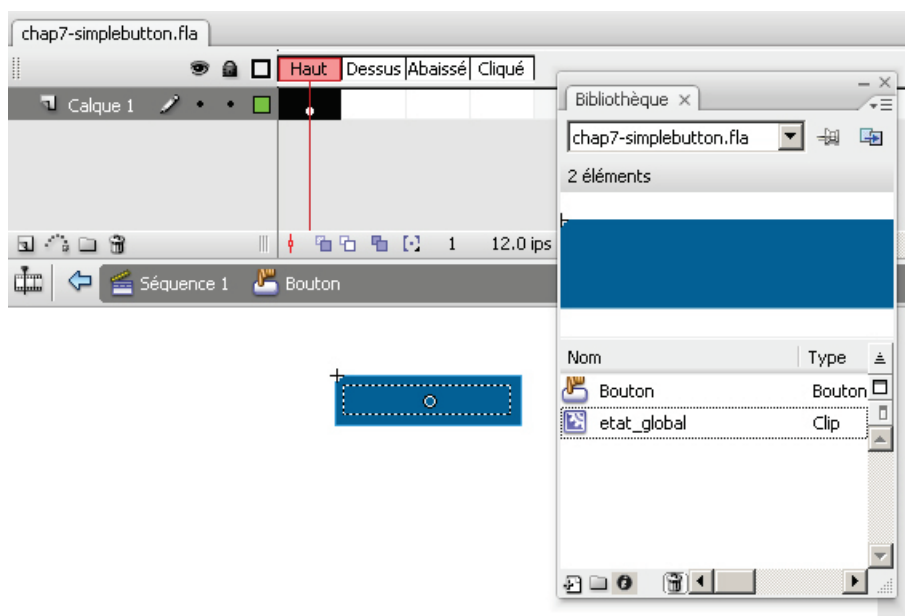
La figure 7-4 illustre le résultat :



*Figure 7-4. Bouton avec légende comme état Haut.*

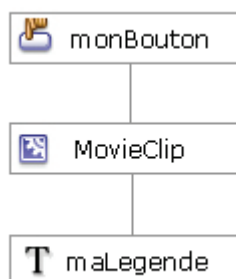
Lorsque nous survolons le bouton, les trois autres états sont conservés seul l'état `upState` (Haut) a été écrasé. Ce comportement nous apprend que même si un seul état a été défini depuis l'environnement auteur, le lecteur copie l'état Haut pour chaque état du bouton. Si nous altérons un état les autres continuent de fonctionner.

Ce n'est pas le résultat que nous avions escompté, il va nous falloir utiliser une autre technique. Le problème vient du fait que nous n'ajoutons pas le champ texte à un objet servant d'état mais nous remplaçons un état par un champ texte. Afin d'obtenir ce que nous souhaitons nous allons depuis l'environnement auteur convertir l'état Haut en clip et y imbriquer un champ texte auquel nous donnons `maLegende` comme nom d'occurrence.



*Figure 7-4. Clip avec champ texte imbriqué pour l'état Haut.*

Nous devons obtenir un clip positionné sur l'état Haut, avec un champ texte imbriqué comme l'illustre la figure 7-5.



*Figure 7-5. Liste d'affichage du bouton.*

Si nous ciblons l'état `upState` de notre occurrence, nous récupérons notre clip tout juste créé :

```
// récupération du clip positionné pour l'état Haut
// affiche : [object MovieClip]
trace( monBouton.upState );
```

Puis nous ciblons le champ texte par la syntaxe pointée traditionnelle :

```
// variable référençant le clip utilisé pour l'état Haut
var etatHaut:MovieClip = MovieClip ( monBouton.upState );

// affectation de contenu
etatHaut.maLegende.text = "Ma légende";
```

Nous pourrions être tentés de donner un nom d'occurrence au clip positionné sur l'état haut, mais souvenons-nous que seules les propriétés `upState`, `downState`, `overState` et `hitTestState` permettent d'accéder aux différents états.

Même si notre clip imbriqué s'appelait `monClipImbrique` le code suivant renverrait `undefined` :

```
// affiche : undefined
trace( monBouton.monClipImbrique );
```

Par défaut le lecteur Flash duplique l'état Haut pour chaque état à la compilation. Ce qui garantit que lorsqu'un état vient à être modifié à l'exécution, comme pour l'affectation de contenu au sein du champ texte, les autres états ne reflètent pas la modification.

Lorsque nous souhaitons avoir un seul état global au bouton, nous trompons le lecteur en affectant notre clip servant d'état Haut à chaque état :

```
// variable référençant le clip utilisé pour l'état Haut
var etatHaut:MovieClip = MovieClip ( monBouton.upState );

// affectation de contenu
etatHaut.maLegende.text = "Ma légende";

//affectation du clip pour tous les états
monBouton.upState = etatHaut;
monBouton.downState = etatHaut;
monBouton.overState = etatHaut;
monBouton.hitTestState = etatHaut;
```

Nous obtenons le résultat suivant :

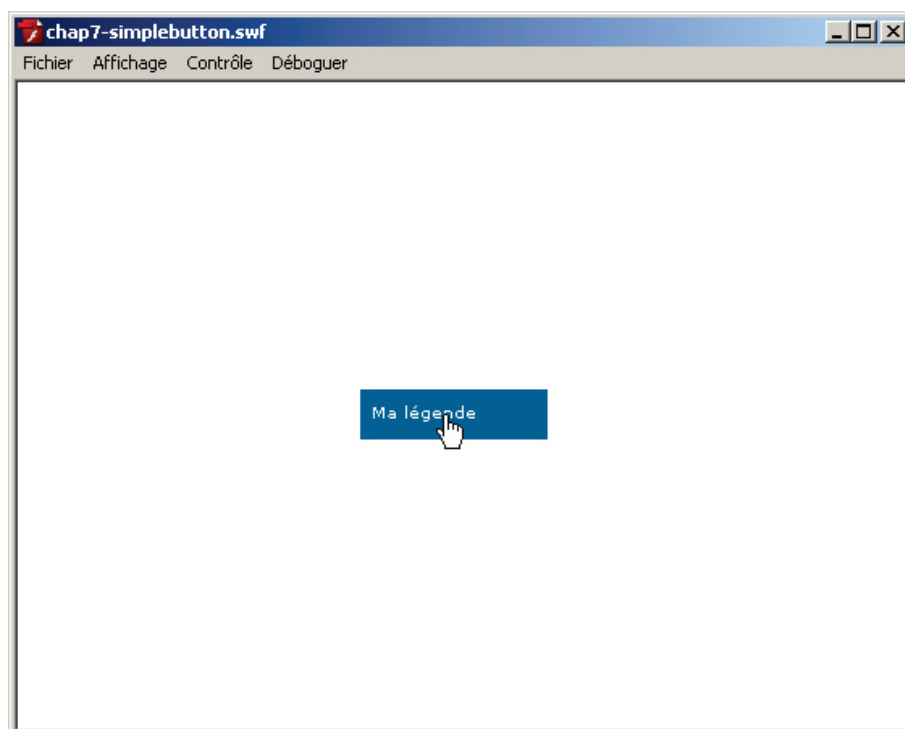


Figure 7-6. Occurrence de *SimpleButton* décoré.

Une fois notre symbole bouton décoré, passons maintenant à son intégration au sein d'un menu.

### A retenir

- La classe `SimpleButton` est une classe particulière et n'hérite pas de la classe `DisplayObjectContainer`.
- Chaque état d'un `SimpleButton` est défini par quatre propriétés : `upState`, `overState`, `downState` et `hitTestState`.

## Créer un menu dynamique

Toute application ou site internet Flash se doit d'intégrer une interface de navigation permettant à l'utilisateur de naviguer au sein du contenu. Le menu figure parmi les classiques du genre, nous avons tous développé au moins une fois un menu.

La mise en application de ce dernier va s'avérer intéressante afin de découvrir de nouveaux comportements apportés par ActionScript 3. Nous allons ensemble créer un menu dynamique en intégrant le bouton sur lequel nous avons travaillé jusqu'à maintenant.

Notre menu sera déployé verticalement, nousinstancions chaque occurrence à travers une boucle `for` :

```
// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void
{
    var lng:int = 5;
    var monBouton:Bouton;

    for ( var i:int = 0; i< lng; i++ )
    {
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

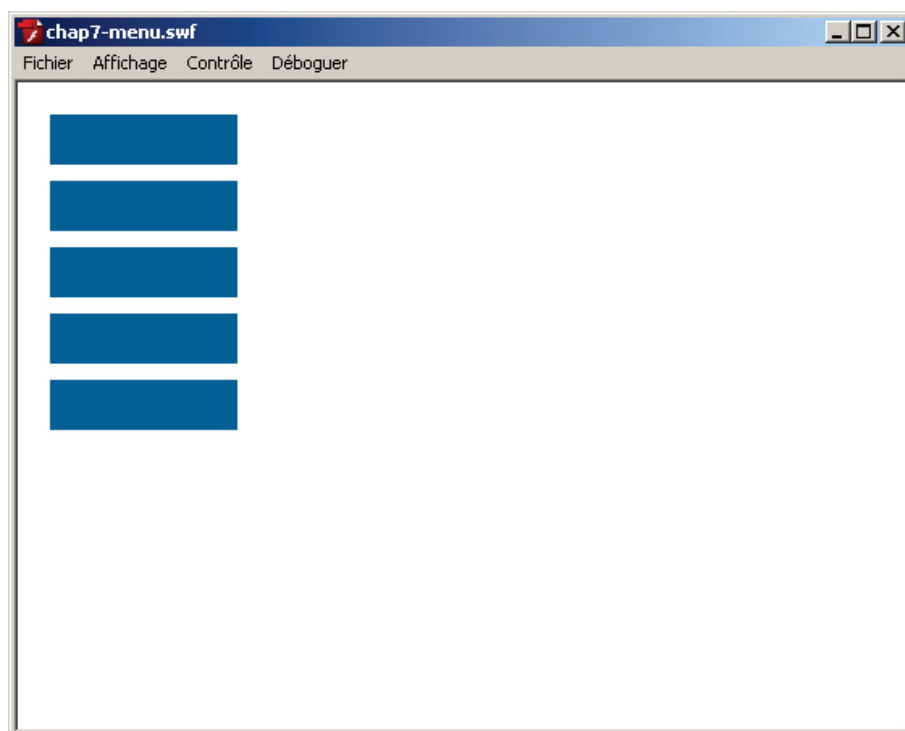
        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut;
        monBouton.overState = etatHaut;
        monBouton.hitTestState = etatHaut;

        // disposition des instances
        monBouton.y = 20 + i * (monBouton.height + 10);

        conteneur.addChild ( monBouton );
    }
}

creeMenu();
```

En testant notre code, nous obtenons le résultat illustré par la figure suivante :



*Figure 7-7. Instances de symboles Bouton.*

Durant la boucle `for` nous décalons chaque bouton du menu grâce à la valeur de la variable `i` qui est incrémentée. En multipliant la hauteur de chaque occurrence par `i` nous obtenons une position `y` spécifique à chaque bouton.

Très souvent, un menu est généré dynamiquement à partir de données externes provenant d'un tableau local, d'un fichier XML local, ou de données provenant d'un serveur. Nous allons modifier le code précédent en définissant un tableau contenant le nom des rubriques à représenter au sein du menu.

Le nombre de boutons du menu sera lié au nombre d'éléments du tableau :

```
// les rubriques
var legendes:Array = new Array ( "Accueil", "Photos", "Liens", "Contact" );

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void
{
```

```
var lng:int = legendes.length;

var monBouton:Bouton;

for ( var i:int = 0; i< lng; i++ )
{
    // création des occurrences du symbole Bouton
    monBouton = new Bouton();

    // variable référençant le clip utilisé pour l'état Haut
    var etatHaut:MovieClip = MovieClip ( monBouton.upState );

    //affectation du clip pour tous les états
    monBouton.upState = etatHaut;
    monBouton.downState = etatHaut;
    monBouton.overState = etatHaut;
    monBouton.hitTestState = etatHaut;

    // disposition des instances
    monBouton.y = 20 + i * (monBouton.height + 10);

    conteneur.addChild ( monBouton );
}
}

creeMenu();
```

Notre menu est maintenant lié au nombre d'éléments du menu, si nous retirons ou ajoutons des éléments au tableau source de données, notre menu sera mis à jour automatiquement. Afin que chaque bouton affiche une légende spécifique nous récupérerons chaque valeur contenue dans le tableau et l'affectons à chaque champ texte contenu dans les boutons.

Nous pouvons facilement récupérer le contenu du tableau au sein de la boucle, grâce à la syntaxe crochet :

```
function creeMenu ():void
{
    var lng:int = legendes.length;

    var monBouton:Bouton;

    for ( var i:int = 0; i< lng; i++ )
    {
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        /* affiche :
        Accueil
        Photos
        Liens
```



```
Contact
*/
trace( legendes[i] );

// variable référençant le clip utilisé pour l'état Haut
var etatHaut:MovieClip = MovieClip ( monBouton.upState );

//affectation du clip pour tous les états
monBouton.upState = etatHaut;
monBouton.downState = etatHaut;
monBouton.overState = etatHaut;
monBouton.hitTestState = etatHaut;

// disposition des instances
monBouton.y = 20 + i * (monBouton.height + 10);

conteneur.addChild ( monBouton );

}

}
```

Au sein de la boucle nous ciblons notre champ texte et nous lui affectons le contenu :

```
function creeMenu ():void
{
    var lng:int = legendes.length;

    var monBouton:Bouton;

    for ( var i:int = 0; i< lng; i++ )
    {
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        // affectation du contenu
etatHaut.maLegende.text = legendes[i];

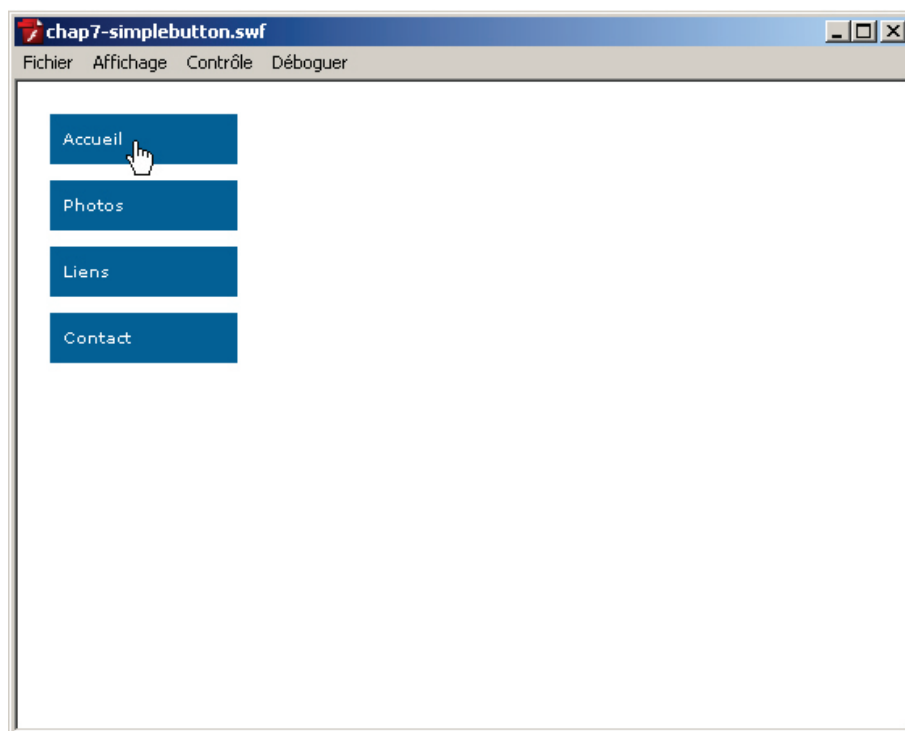
        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut;
        monBouton.overState = etatHaut;
        monBouton.hitTestState = etatHaut;

        // disposition des instances
        monBouton.y = 20 + i * (monBouton.height + 10);

        conteneur.addChild ( monBouton );

    }
}
```

En testant le code précédent nous obtenons le résultat illustré par la figure 7-8 :



*Figure 7-8. Menu dynamique avec légendes.*

Notre menu est terminé, nous pourrions en rester là mais il faut avouer que ce dernier manque de vie. Nous allons lui donner vie en ajoutant un peu de mouvement.

A l'aide d'un effet de glissement, nous allons rendre ce dernier plus attrayant. En route vers la notion de mouvement programmatique !

## Mouvement programmatique

Afin de créer différents mouvements par programmation nous pouvons utiliser nos propres algorithmes, différentes bibliothèques open-source ou bien la classe `Tween` intégrée à Flash CS3.

La classe `Tween` réside dans le paquetage `fl.transitions` et doit être importée afin d'être utilisée. Afin de donner du mouvement à un objet graphique dans Flash, nous pouvons utiliser les événements `Event.ENTER_FRAME` ou `TimerEvent.TIMER`.

La classe `Tween` utilise en interne un événement `Event.ENTER_FRAME` afin de modifier la propriété de l'objet. Un large nombre de mouvements sont disponibles, de type cinétique, élastique, rebond ou bien constant.

Nous allons ajouter un effet d'élasticité permettant à notre menu d'être déployé de manière ludique. Pour cela nous associons un objet `Tween` à chaque instance de bouton. Chaque référence à l'objet `Tween` est stockée au sein de l'occurrence bouton pour pouvoir être récupérée facilement plus tard.

---

La classe `Tween` n'est pas automatiquement importée pour le compilateur, nous devons donc le faire afin de l'utiliser.

---

Nous écoutons l'événement `MouseEvent.CLICK` des boutons :

```
// import des classes Tween et Elastic pour le type de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;

// les rubriques
var legendes:Array = new Array ( "Accueil", "Photos", "Liens", "Contact" );

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void
{
    var lng:int = legendes.length;

    var monBouton:Bouton;

    for (var i:int = 0; i< lng; i++ )
    {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        etatHaut.maLegende.text = legendes[i];

        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut;
        monBouton.overState = etatHaut;
        monBouton.hitTestState = etatHaut;

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
i * (monBouton.height + 10), 3, true );

        conteneur.addChild ( monBouton );
    }
}
```

```
    }  
  }  
  creeMenu();  
  
  // capture de l'événement MouseEvent.CLICK auprès du conteneur  
  conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );  
  
  function clicMenu ( pEvt:MouseEvent ):void  
  {  
  
    // affiche : [object Bouton]  
    trace( pEvt.target );  
  
    // affiche : [object Sprite]  
    trace( pEvt.currentTarget );  
  
  }  
}
```

La position dans l'axe des y de chaque bouton est désormais gérée par notre objet **Tween**. En stockant chaque objet **Tween** créé au sein des boutons nous pourrons y faire référence à n'importe quel moment en ciblant plus tard la propriété **tween**.

Notre menu se déploie avec un effet d'élasticité et devient beaucoup plus interactif. Nous ne sommes pas restreints à un seul type de mouvement, nous allons aller plus loin en ajoutant un effet similaire lors du survol. Cette fois, le mouvement se fera sur la largeur des boutons.

Nous stockons une nouvelle instance de la classe **Tween** pour gérer l'effet de survol de chaque bouton :

```
// import des classes Tween et Elastic pour le type de mouvement  
import fl.transitions.Tween;  
import fl.transitions.easing.Elastic;  
// les rubriques  
var legendes:Array = new Array ( "Accueil", "Photos", "Liens", "Contact" );  
  
// création du conteneur  
var conteneur:Sprite = new Sprite();  
  
conteneur.x = 20;  
  
addChild ( conteneur );  
  
function creeMenu ():void  
{  
  
  var lng:int = legendes.length;  
  
  var monBouton:Bouton;  
  
  for ( var i:int = 0; i< lng; i++ )  
  
  {
```

```
// création des occurrences du symbole Bouton
monBouton = new Bouton();

// variable référençant le clip utilisé pour l'état Haut
var etatHaut:MovieClip = MovieClip ( monBouton.upState );

etatHaut.maLegende.text = legendes[i];

//affectation du clip pour tous les états
monBouton.upState = etatHaut;
monBouton.downState = etatHaut;
monBouton.overState = etatHaut;
monBouton.hitTestState = etatHaut;

// disposition des instances
monBouton.tween = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
i * (monBouton.height + 10), 3, true );

    // un objet Tween est créé pour les effets de survol
    monBouton.tweenSurvol = new Tween ( monBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

    conteneur.addChild ( monBouton );

}

}

creeMenu();

// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );
conteneur.addEventListener ( MouseEvent.ROLL_OVER, survolBouton, true );
conteneur.addEventListener ( MouseEvent.ROLL_OUT, quitteBouton, true );

function survolBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;

    monTween.continueTo ( 1.1, 2 );
}

function quitteBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;

    monTween.continueTo ( 1, 2 );
}

function clicMenu ( pEvt:MouseEvent ):void
{
    // affiche : [object Bouton]
    trace( pEvt.target );
```

```
| // affiche : [object Sprite]
| trace( pEvt.currentTarget );
| }
```

Au sein des fonctions `survolBouton` et `quitteBouton` nous récupérerons l'objet `Tween` associé à l'objet survolé au sein de la variable `monTween`. Grâce à la méthode `continueTo` de l'objet `Tween`, nous pouvons redémarrer, stopper ou bien lancer l'animation dans le sens inverse. Dans le code précédent nous avons défini les valeurs de départ et d'arrivée pour l'étirement du bouton. Nous augmentons de 10% la taille du bouton au survol et nous ramenons sa taille à 100% lorsque nous quittons le survol du bouton.

Nous obtenons un menu élastique réagissant au survol, mais il nous reste une chose à optimiser. Si nous regardons bien, nous voyons que le champ texte interne au bouton est lui aussi redimensionné. Ce problème intervient car nous redimensionnons l'enveloppe principale du bouton dans lequel se trouve notre champ texte. En étirant l'enveloppe conteneur nous étirons les enfants et donc la légende.

Nous allons modifier notre symbole `Bouton` afin de pouvoir redimensionner la forme de fond de notre bouton sans altérer le champ texte.

Pour cela, au sein du symbole `Bouton` nous éditons le clip placé sur l'état Haut et transformons la forme de fond en clip auquel nous donnons `fondBouton` comme nom d'occurrence. Nous allons ainsi redimensionner le clip interne à l'état `upState`, sans altérer le champ texte. Nous modifions notre code en associant l'objet `Tween` au clip `fondBouton` :

```
| // un objet Tween est créé pour les effets de survol
| monBouton.tweenSurvol = new Tween ( etatHaut.fondBouton, "scaleX",
| Elastic.easeOut, 1, 1, 2, true );
```

Au final, la classe `SimpleButton` s'avère très pratique mais ne facilite pas tellement la tâche dès lors que nos boutons sont quelque peu travaillés. Contrairement à la classe `Button` en ActionScript 1 et 2, la classe `SimpleButton` peut diffuser un événement `Event.ENTER_FRAME`. L'utilisation de la classe `SimpleButton` s'avère limitée de par son manque de souplesse en matière de décoration.

Nous allons refaire le même exercice en utilisant cette fois des instances de `Sprite` auxquels nous ajouterons un comportement bouton, une fois terminé nous ferons un bilan.

## A retenir

- La classe `Tween` permet d'ajouter du mouvement à nos objets graphiques.
- Celle-ci réside dans le paquetage `fl.transitions` et doit être importée explicitement afin d'être utilisée.
- Les différentes méthodes et propriétés de la classe `Tween` permettent de gérer le mouvement.

## Les pièges du ramasse-miettes

Comme nous l'avons vu depuis le début de l'ouvrage, le ramasse-miettes figure parmi les éléments essentiels à prendre en considération lors du développement d'applications `ActionScript 3`.

Dans l'exemple précédent, nous avons utilisé la classe `Tween` afin de gérer les mouvements de chaque bouton. Une ancienne habitude provenant des précédentes versions d'`ActionScript` pourrait nous pousser à ne pas conserver de références aux objets `Tween`.

Dans le code suivant, nous avons modifié la fonction `creeMenu` de manière à ne pas stocker de références aux objets `Tween` :

```
function creeMenu ():void
{
    var lng:int = legendes.length;

    var monBouton:Bouton;
    var tweenMouvement:Tween;
    var tweenSurvol:Tween;

    for ( var i:int = 0; i< lng; i++ )
    {
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // variable référençant le clip utilisé pour l'état Haut
        var etatHaut:MovieClip = MovieClip ( monBouton.upState );

        etatHaut.maLegende.text = legendes[i];

        //affectation du clip pour tous les états
        monBouton.upState = etatHaut;
        monBouton.downState = etatHaut;
        monBouton.overState = etatHaut;
        monBouton.hitTestState = etatHaut;

        // disposition des instances
        tweenMouvement = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
        i * (monBouton.height + 10), 3, true );
    }
}
```

```
// un objet Tween est créé pour les effets de survol
tweenSurvol = new Tween ( etatHaut.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

conteneur.addChild ( monBouton );

}

}
```

Une fois l'exécution de la fonction `creeMenu`, les variables `tweenMouvement` et `tweenSurvol` sont supprimées. Nos objets `Tween` ne sont plus référencés au sein de notre application.

Si nous déclenchons manuellement le passage du ramasse-miettes après la création du menu :

```
creeMenu();

// déclenchement du ramasse-miettes
System.gc();
```

Nous remarquons que le mouvement de chaque bouton est interrompu, car le ramasse-miettes vient de supprimer les objets `Tween` de la mémoire.

## A retenir

- Veillez à bien référencer les objets nécessaires afin qu'ils ne soient pas supprimés par le ramasse-miettes sans que vous ne l'ayez décidé.

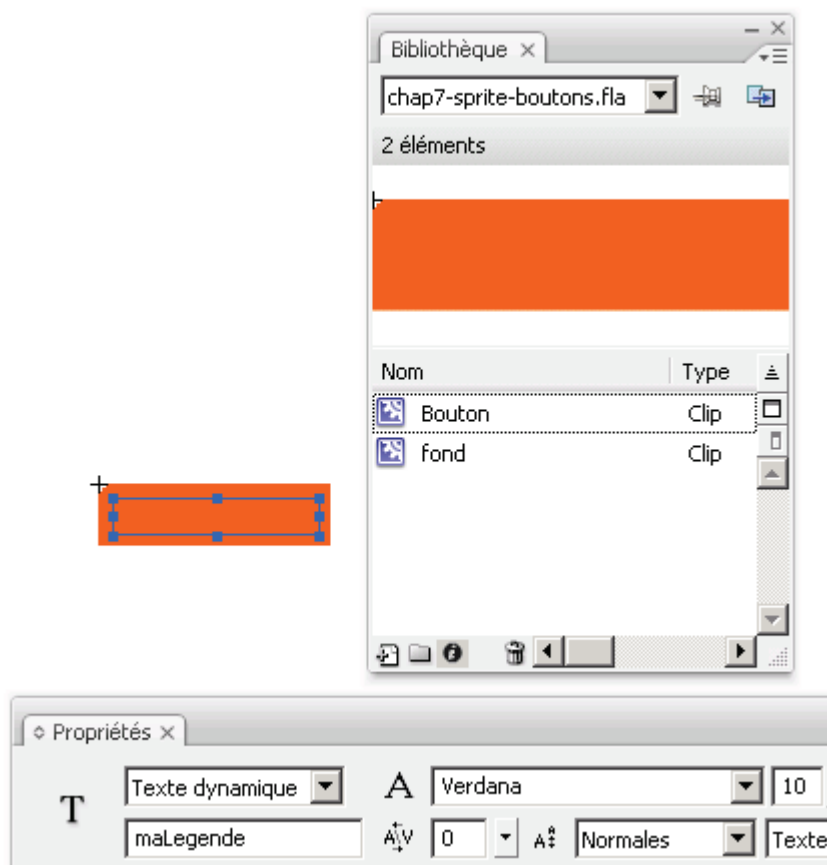
## Ajout de comportement bouton

Un grand nombre de développeurs Flash utilisaient en ActionScript 1 et 2 des symboles clips en tant que boutons. En définissant l'événement `onRelease` ces derniers devenaient cliquables. En ActionScript 3 le même comportement peut être obtenu en activant la propriété `buttonMode` sur une occurrence de `Sprite` ou `MovieClip`.

Dans un nouveau document, nous créons un symbole `Sprite` grâce à la technique abordée lors du chapitre 5 intitulé *Les symboles*. Nous lui associons `Bouton` comme nom de classe grâce au panneau *Propriétés de Liaison*, puis nous transformons la forme contenue dans ce dernier en un nouveau clip.

Nous lui donnons `fondBouton` comme nom d'occurrence. Au dessus de ce clip nous créons un champ texte dynamique que nous appelons `maLegende` :





*Figure 7-9. Symbole clip.*

Puis nous disposons nos instances de **Bouton** verticalement :

```
// les rubriques
var legendes:Array = new Array ( "Accueil", "Nouveautés", "Photos", "Liens",
"Contact" );

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

function creeMenu ():void
{
    // nombre de rubriques
    var lng:int = legendes.length;
    var monBouton:Bouton;

    for ( var i:int = 0; i< lng; i++ )
    {
```

```
// instantiation du symbole Bouton
monBouton = new Bouton();

monBouton.y = 20 + i * (monBouton.height + 10);

// ajout à la liste d'affichage
conteneur.addChild ( monBouton );

}

}

creeMenu();
```

Afin d'activer le comportement bouton sur des objets autres que `SimpleButton` nous devons activer la propriété `buttonMode` :

```
// activation du comportement bouton
monBouton.buttonMode = true;
```

Puis nous ajoutons le texte :

```
function creeMenu ():void
{
    // nombre de rubriques
    var lng:int = legendes.length;
    var monBouton:Bouton;

    for ( var i:int = 0; i < lng; i++ )
    {
        // instantiation du symbole Bouton
        monBouton = new Bouton();

        monBouton.y = 20 + i * (monBouton.height + 10);

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // affectation du contenu
        monBouton.maLegende.text = legendes[i];

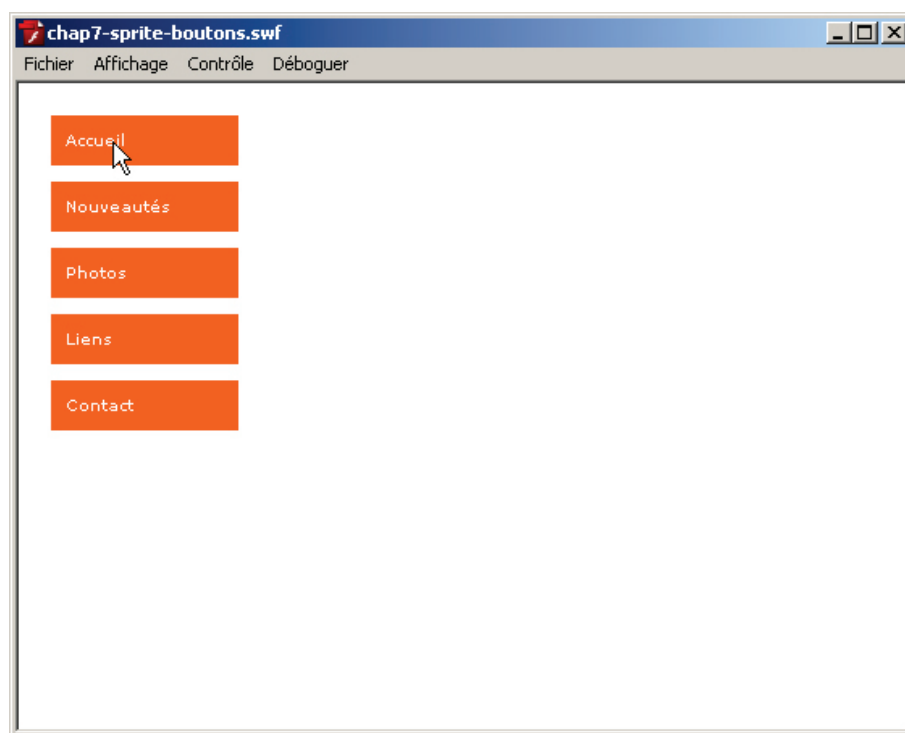
        // ajout à la liste d'affichage
        conteneur.addChild ( monBouton );
    }
}
```

En utilisant d'autres objets que la classe `SimpleButton` pour créer des boutons nous devons prendre en considération certains comportements comme la réactivité des objets imbriqués avec la souris. La manière dont les objets réagissent aux événements souris a été modifiée en ActionScript 3. Nous allons à présent découvrir ces subtilités.

## Zone réactive

Si nous survolons les boutons de notre menu, nous remarquons que le curseur représentant une main ne s’affiche pas sur la totalité de la surface des boutons.

Le comportement est illustré par la figure suivante :



*Figure 7-10. Zone réactive.*

Si nous cliquons sur la zone occupée par le champ texte imbriqué dans chaque bouton, l’objet cible n’est plus le bouton mais le champ texte. A l’inverse si nous cliquons sur la zone non occupée par le champ texte, l’objet cible est le clip `fondBouton`. Ainsi, en cliquant sur les boutons de notre menu, l’objet réactif peut être le clip ou le champ texte imbriqué.

Nous capturons l’événement `MouseEvent.CLICK` auprès du conteneur :

```
// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );

function clicMenu ( pEvt:MouseEvent ):void
{
    // affiche : [object TextField]
    trace( pEvt.target );
}
```

```
| // affiche : [object Sprite]
| trace( pEvt.currentTarget );
| }
```

A chaque clic la fonction `clicMenu` est déclenchée, la propriété `target` de l'objet événementiel renvoie une référence vers l'objet cible de l'événement. La propriété `currentTarget` référence le conteneur actuellement notifié de l'événement `MouseEvent.CLICK`.

---

Souvenez-vous, la propriété `target` référence l'objet cible de l'événement. La propriété `currentTarget` référence l'objet sur lequel nous avons appelé la méthode `addEventListener`.

---

Si nous cliquons sur le bouton, les objets enfants réagissent aux entrées souris. La propriété `target` renvoie une référence vers chaque objet interactif. Si nous cliquons à côté du champ texte imbriqué, le clip `fondBouton` réagit :

```
| function clicMenu ( pEvt:MouseEvent ):void
| {
|     // affiche : [object MovieClip]
|     trace( pEvt.target );
| }
```

Si nous cliquons sur le champ texte imbriqué la propriété `target` renvoie une référence vers ce dernier :

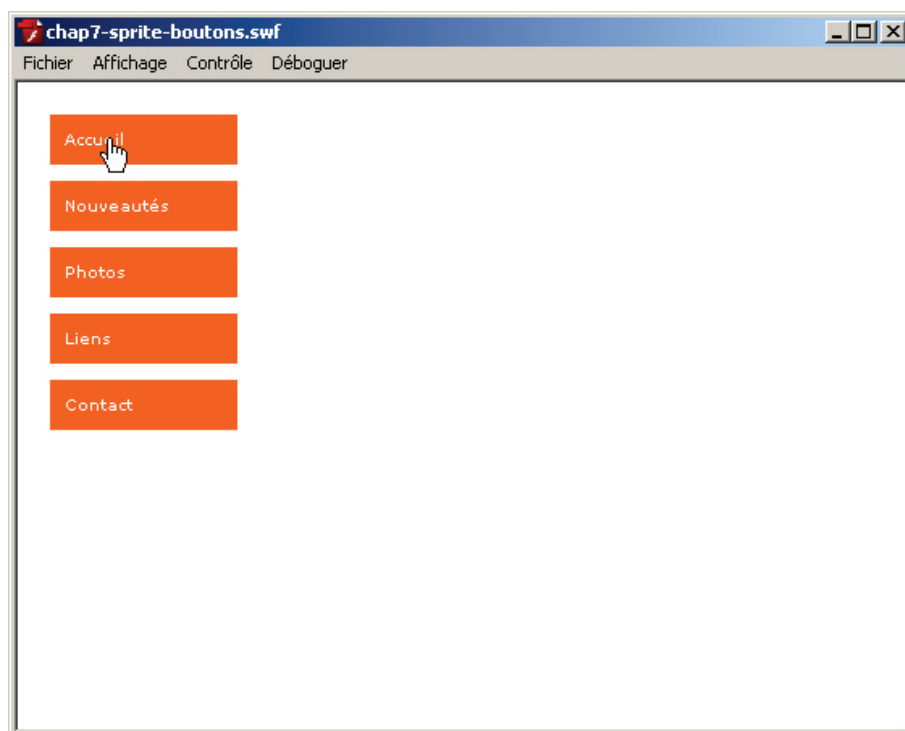
```
| function clicMenu ( pEvt:MouseEvent ):void
| {
|     // affiche : [object TextField]
|     trace( pEvt.target );
| }
```

Afin d'être sûrs que notre zone sensible sera uniquement l'enveloppe parente nous désactivons la sensibilité à la souris pour tous les objets enfants grâce à la propriété `mouseChildren`.

Ainsi, l'objet cible sera l'enveloppe principale du bouton :

```
| // désactivation des objets enfants
| monBouton.mouseChildren = false;
```

Une fois la propriété `mouseChildren` passée à `false`, l'ensemble des objets enfants au bouton ne réagissent plus à la souris. Notre bouton est parfaitement cliquable sur toute sa surface :



*Figure 7-11. Désactivation des objets enfants.*

En cliquant sur chacun des boutons, la propriété `target` référence le bouton cliqué :

```
function clicMenu ( pEvt:MouseEvent ):void
{
    // affiche : [object Bouton]
    trace( pEvt.target );
}
```

Les boutons du menu sont désormais parfaitement cliquables, nous allons intégrer la classe `Tween` que nous avons utilisé dans l'exemple précédent :

```
// import des classes Tween et Elastic pour le type de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;

// création du conteneur
var conteneur:Sprite = new Sprite();

conteneur.x = 20;

addChild ( conteneur );

// les rubriques
var legendes:Array = new Array ( "Accueil", "Nouveautés", "Photos", "Liens",
    "Contact" );
```

```
function creeMenu ()
{
    var lng:int = legendes.length;
    var monBouton:Bouton;
    for ( var i:int = 0; i< lng; i++ )
    {
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = legendes[i];

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "y", Elastic.easeOut, 0, 20 +
i * (monBouton.height + 10), 3, true );

        // un objet Tween est créé pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );
    }
}

creeMenu();

// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );
conteneur.addEventListener ( MouseEvent.ROLL_OVER, survolBouton, true );
conteneur.addEventListener ( MouseEvent.ROLL_OUT, quitteBouton, true );

function survolBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;
    monTween.continueTo ( 1.1, 2 );
}

function quitteBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;
    monTween.continueTo ( 1, 2 );
}
```

```
}

function clicMenu ( pEvt:MouseEvent ):void
{
    // affiche : [object Bouton]
    trace( pEvt.target );

    // affiche : [object Sprite]
    trace( pEvt.currentTarget );
}
```

Nous obtenons le même menu qu’auparavant à l’aide d’occurrences de `SimpleButton`. Si nous souhaitons donner un style différent à notre menu, nous pouvons jouer avec les propriétés et méthodes des objets `Tween`.

Dans le code suivant nous disposons le menu avec un effet de rotation. Pour cela nous modifions simplement les lignes gérant la disposition des occurrences :

```
function creeMenu ()
{
    var lng:int = legendes.length;

    var monBouton:Bouton;
    var angle:int = 360 / lng;

    for ( var i:int = 0; i < lng; i++ )
    {
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = legendes[i];

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

        // un objet Tween est créé pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );
    }
}
```

```
| }

```

Puis nous déplaçons le conteneur :

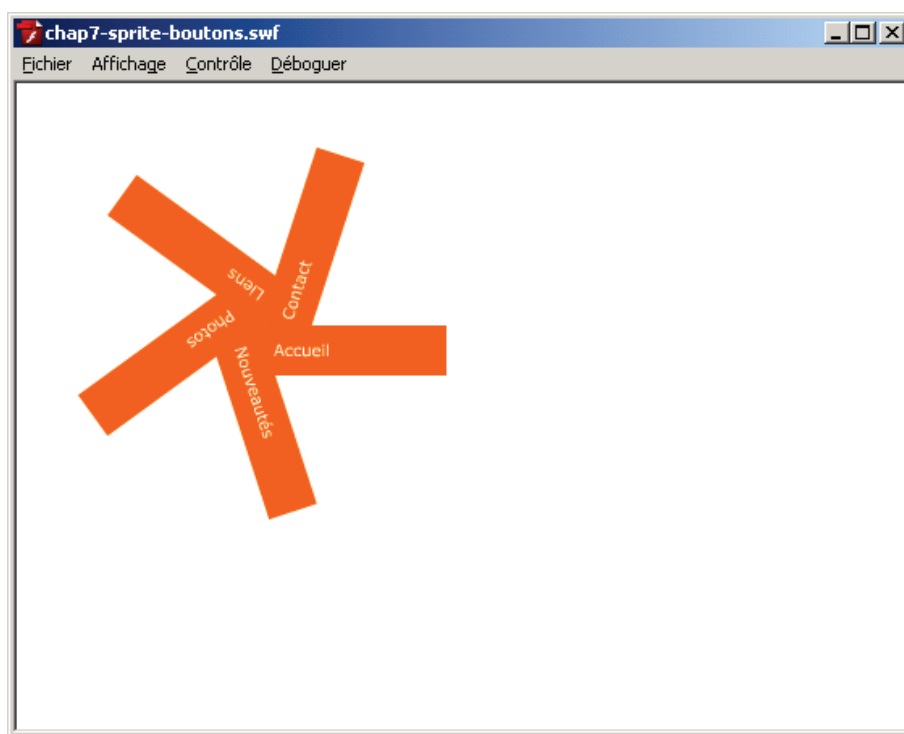
```
| conteneur.x = 150;
| conteneur.y = 150;

```

Si le texte des boutons disparaît, cela signifie que nous n'avons pas intégré les contours de polices pour nos champs texte.

Le lecteur Flash ne peut rendre à l'affichage un champ texte ayant subi une rotation ou un masque, s'il contient une police non embarquée. Il faut toujours s'assurer d'avoir bien intégré les contours de polices.

Une fois les contours de polices intégrés, en modifiant simplement ces quelques lignes nous obtenons un menu totalement différent comme l'illustre la figure 7-12 :



*Figure 7-12. Disposition du menu avec effet de rotation.*

Libre à vous d'imaginer toutes sortes d'effets en travaillant avec les différentes propriétés de la classe `DisplayObject`. Dans le code précédent nous avons modifié la propriété `rotation`.

Une application Flash peut prendre en compte des comportements interactifs relativement subtils comme la perte de focus de



l'animation. Le lecteur Flash 9 peut en ActionScript 3 détecter facilement la perte et le gain du focus de l'animation. Nous allons adapter notre menu afin qu'il prenne en considération cette fonctionnalité.

## A retenir

- La propriété `buttonMode` affecte un comportement bouton aux objets `MovieClip` et `Sprite`.
- Pour s'assurer que l'enveloppe principale seulement reçoive les entrées souris, nous passons la valeur `false` à la propriété `mouseChildren`.

## Gestion du focus

Nous allons travailler sur la gestion du focus à présent, deux nouveaux événements ont vu le jour en ActionScript 3 :

- `Event.ACTIVATE` : événement diffusé lorsque le lecteur Flash gagne le focus.
- `Event.DEACTIVATE` : événement diffusé lorsque le lecteur Flash perd le focus.

Ces deux événements sont diffusés par tout `DisplayObject` présent ou non au sein de la liste d'affichage. Grâce à ces événements nous allons pouvoir fermer le menu lorsque l'animation perdra le focus puis le rouvrir à l'inverse.

Pour savoir quand l'utilisateur n'a plus le focus sur le lecteur il suffit d'écouter l'événement `Event.DEACTIVATE` de n'importe quel `DisplayObject`, qu'il soit sur la liste d'affichage ou non :

```
// souscription auprès de l'événement Event.DEACTIVATE auprès du conteneur
conteneur.addEventListener ( Event.DEACTIVATE, perteFocus );

function perteFocus ( pEvt:Event ):void
{
    // récupération du nombre de boutons
    var lng:int = pEvt.target.numChildren;
    var bouton:Bouton;

    for (var i:int = 0; i< lng; i++ )
    {
        // nous récupérons chaque bouton du menu
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
```

```
        myTween.func = Strong.easeOut;
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( i * 10, 1 );
    }
}
```

Attention, la classe **Strong** doit être importée :

```
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;
import fl.transitions.easing.Strong;
```

Lorsque l’animation perd le focus notre menu se referme avec un effet d’inertie, la figure suivante illustre le résultat :



*Figure 7-13. Menu refermé.*

Il nous faut ouvrir à nouveau le menu lorsque l’application récupère le focus, pour cela nous écoutons l’événement **Event.ACTIVATE** :

```
function perteFocus ( pEvt:Event ):void
{
    // récupération du nombre de boutons
    var lng:int = pEvt.target.numChildren;
    var bouton:Bouton;

    for (var i:int = 0; i < lng; i++ )
    {
        // nous récupérons chaque bouton du menu
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
        myTween.func = Strong.easeOut;
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( i * 10, 1 );
    }

    if ( pEvt.target.hasEventListener( Event.ACTIVATE ) == false )
```

```
{  
  
    // souscription auprès de l'événement Event.ACTIVATE auprès du  
    conteneur  
    pEvt.target.addEventListener ( Event.ACTIVATE, gainFocus );  
  
}  
  
}  
  
function gainFocus ( pEvt:Event ):void  
{  
  
    // récupération du nombre de boutons  
    var lng:int = pEvt.target.numChildren;  
    var bouton:Bouton;  
  
    for (var i:int = 0; i< lng; i++ )  
  
    {  
  
        // nous récupérons chaque bouton du menu  
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );  
  
        // nous ciblons chaque objet Tween  
        var myTween:Tween = bouton.tween;  
        myTween.func = Elastic.easeOut;  
        // nous définissons les points de départ et d'arrivée  
        myTween.continueTo ( (360/lng) * i, 2 );  
  
    }  
  
}
```

Nous déclenchons les différents mouvements l'aide la méthode `continueTo` de l'objet `Tween`, nous pouvons jouer avec les différentes valeurs et propriétés altérées par le mouvement pour obtenir d'autres effets :

```
Voici le code complet de notre menu dynamique :  
// import des classes Tween et Elastic pour le type de mouvement  
import fl.transitions.Tween;  
import fl.transitions.easing.Elastic;  
import fl.transitions.easing.Strong;  
  
// création du conteneur  
var conteneur:Sprite = new Sprite();  
  
conteneur.x = 150;  
conteneur.y = 150;  
  
addChild ( conteneur );  
  
// les rubriques  
var legendes:Array = new Array ( "Accueil", "Nouveautés", "Photos", "Liens",  
    "Contact" );  
  
function creeMenu ()  
{
```

```
var lng:int = legendes.length;

var monBouton:Bouton;
var angle:int = 360 / lng;

for ( var i:int = 0; i< lng; i++ )
{
    // création des occurrences du symbole Bouton
    monBouton = new Bouton();

    // activation du comportement bouton
    monBouton.buttonMode = true;

    // désactivation des objets enfants
    monBouton.mouseChildren = false;

    // affectation du contenu
    monBouton.maLegende.text = legendes[i];

    // disposition des instances
    monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

    // un objet Tween est créé pour les effets de survol
    monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

    conteneur.addChild ( monBouton );
}
}

creeMenu();

// capture de l'événement click sur le scénario principal
conteneur.addEventListener ( MouseEvent.CLICK, clicMenu, true );
conteneur.addEventListener ( MouseEvent.ROLL_OVER, survolBouton, true );
conteneur.addEventListener ( MouseEvent.ROLL_OUT, quitteBouton, true );

function survolBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;

    monTween.continueTo ( 1.1, 2 );
}

function quitteBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;

    monTween.continueTo ( 1, 2 );
}
```

```

function clicMenu ( pEvt:MouseEvent ):void
{
    // affiche : [object Bouton]
    trace( pEvt.target );

    // affiche : [object Sprite]
    trace( pEvt.currentTarget );
}

// souscription auprès de l'événement Event.DEACTIVATE auprès du conteneur
conteneur.addEventListener ( Event.DEACTIVATE, perteFocus );

function perteFocus ( pEvt:Event ):void
{
    // récupération du nombre de boutons
    var lng:int = pEvt.target.numChildren;
    var bouton:Bouton;

    for (var i:int = 0; i< lng; i++ )
    {
        // nous récupérons chaque bouton du menu
        bouton = Bouton ( pEvt.target.getChildAt ( i ) );

        // nous ciblons chaque objet Tween
        var myTween:Tween = bouton.tween;
        myTween.func = Strong.easeOut;
        // nous définissons les points de départ et d'arrivée
        myTween.continueTo ( i * 10, 1 );
    }

    if( pEvt.target.hasEventListener( Event.ACTIVATE) == false )
    {
        // souscription auprès de l'événement Event.ACTIVATE auprès du
        conteneur
        pEvt.target.addEventListener ( Event.ACTIVATE, gainFocus );
    }
}

function gainFocus ( pEvt:Event ):void
{
    // récupération du nombre de boutons
    var lng:int = pEvt.target.numChildren;
    var bouton:Bouton;

    for (var i:int = 0; i< lng; i++ )
    {

```

```
// nous récupérons chaque bouton du menu
bouton = Bouton ( pEvt.target.getChildAt ( i ) );

// nous ciblons chaque objet Tween
var myTween:Tween = bouton.tween;
myTween.func = Elastic.easeOut;
// nous définissons les points de départ et d'arrivée
myTween.continueTo ( 60 * (i+1), 2 );

}

}
```

Notre menu dynamique réagit désormais à la perte ou au gain du focus de l'animation. Nous pourrions varier les différents effets et augmenter les capacités du menu sans limites, c'est ici que réside la puissance de Flash !

## A retenir

- Les événements `Event.ACTIVATE` et `Event.DEACTIVATE` permettent de gérer la perte ou le gain de focus du lecteur.

## Pour aller plus loin

Nous n'avons pas encore utilisé la fonction `clicMenu` souscrite auprès de l'événement `MouseEvent.CLICK` de chaque bouton. En associant un lien à chaque bouton nous ouvrirons une fenêtre navigateur.

Il nous faut dans un premier temps stocker chaque lien, pour cela nous allons créer un second tableau spécifique :

```
// les liens
var liens:Array = new Array ("http://www.oreilly.com",
"http://www.bytearray.org", "http://www.flickr.com",
"http://www.linkdup.com", "http://www.myspace.com");
```

Chaque bouton contient au sein de sa propriété `lien` un lien associé :

```
function creeMenu ()
{
    var lng:int = legendes.length;

    var monBouton:Bouton;
    var angle:int = 360 / lng;

    for ( var i:int = 0; i< lng; i++ )
    {

        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
```

```
monBouton.buttonMode = true;

// désactivation des objets enfants
monBouton.mouseChildren = false;

// affectation du contenu
monBouton.maLegende.text = legendes[i];

// chaque bouton stocke son lien associé
monBouton.lien = liens[i];

// disposition des instances
monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

// un objet Tween est créé pour les effets de survol
monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

conteneur.addChild ( monBouton );

}

}
```

Lorsque la fonction `clicMenu` est déclenchée, nous ouvrons une nouvelle fenêtre navigateur :

```
function clicMenu ( pEvt:MouseEvent ):void
{
    // ouvre une fenêtre navigateur pour le lien cliqué
    navigateToURL ( new URLRequest ( pEvt.target.lien ) );
}
```

Nous récupérons le lien au sein du bouton cliqué, référencé par la propriété `target` de l'objet événementiel. La fonction `navigateToURL` stockée au sein du paquetage `flash.net` nous permet d'ouvrir une nouvelle fenêtre navigateur et d'atteindre le lien spécifié. En ActionScript 3 tout lien HTTP doit être enveloppé dans un objet `URLRequest`.

Pour plus d'informations concernant les échanges externes, rendez-vous au chapitre 15 intitulé *Communication externe*.

---

Notons qu'il est possible de créer une propriété `lien` car la classe `MovieClip` est dynamique. La création d'une telle propriété au sein d'une instance de `SimpleButton` est impossible.

---

Notre code pourrait être amélioré en préférant un tableau associatif aux deux tableaux utilisés actuellement. Nous pourrions regrouper nos deux tableaux `liens` et `legendes` en un seul tableau :

```
// rubriques et liens
var donnees:Array = new Array ();

// ajout des rubriques et liens associés
donnees.push ( { rubrique : "Accueil", lien : "http://www.oreilly.com" } );
donnees.push ( { rubrique : "Nouveautés", lien : "http://www.bytearray.org" } );
donnees.push ( { rubrique : "Photos", lien : "http://www.flickr.com" } );
donnees.push ( { rubrique : "Liens", lien : "http://www.linkdup.com" } );
donnees.push ( { rubrique : "Contact", lien : "http://www.myspace.com" } );
```

En utilisant un tableau associatif nous organisons mieux nos données et rendons l'accès simplifié.

Nous modifions la fonction `creeMenu` afin de cibler les informations au sein du tableau associatif :

```
function creeMenu ()
{
    var lng:int = donnees.length;

    var monBouton:Bouton;
    var angle:int = 360 / lng;

    for ( var i:int = 0; i< lng; i++ )
    {
        // création des occurrences du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = donnees[i].rubrique;

        // chaque bouton stocke son lien associé
        monBouton.lien = donnees[i].lien;

        // disposition des instances
        monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, i * angle, 3, true );

        // un objet Tween est créé pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

        conteneur.addChild ( monBouton );
    }
}
```

Notre menu est maintenant terminé ! Intéressons-nous maintenant à l'espace de coordonnées.



## A retenir

- Il est préférable d'utiliser des tableaux associatifs plutôt que des tableaux à index. L'organisation et l'accès aux données seront optimisés et simplifiés.

## Espace de coordonnées

Pour toutes les entrées souris, les objets graphiques diffusent des objets événementiels de type `flash.events.MouseEvent`. Cette classe possède de nombreuses propriétés dont voici le détail :

- `MouseEvent.altKey` : censée indiquer si la touche ALT est enfoncée au moment du clic.
- `MouseEvent.buttonDown` : indique si le bouton principal de la souris est enfoncé au moment du clic.
- `MouseEvent.delta` : indique le nombre de lignes qui doivent défiler chaque fois que l'utilisateur fait tourner la molette de sa souris d'un cran.
- `MouseEvent.localX` : indique les coordonnées X de la souris par rapport à l'espace de coordonnées de l'objet cliqué.
- `MouseEvent.localY` : indique les coordonnées Y de la souris par rapport à l'espace de coordonnées de l'objet cliqué.
- `MouseEvent.relatedObject` : indique l'objet sur lequel la souris pointe lors de l'événement `MouseEvent.MOUSE_OUT`.
- `MouseEvent.shiftKey` : indique si la touche SHIFT est enfoncée au moment du clic.
- `MouseEvent.stageX` : indique les coordonnées X de la souris par rapport à l'espace de coordonnées de l'objet `Stage`.
- `MouseEvent.stageY` : indique les coordonnées Y de la souris par rapport à l'espace de coordonnées de l'objet `Stage`.

En traçant l'objet événementiel, une représentation `toString()` est effectuée :

```
// souscription auprès de l'événement MouseMove du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

function bougeSouris ( pEvt:MouseEvent ):void
{
    //affiche : [MouseEvent type="mouseMove" bubbles=true cancelable=false
    eventPhase=2 localX=28 localY=61 stageX=158.95000000000002 stageY=190
    relatedObject=null ctrlKey=false altKey=false shiftKey=false delta=0]
    trace(pEvt);
}
```

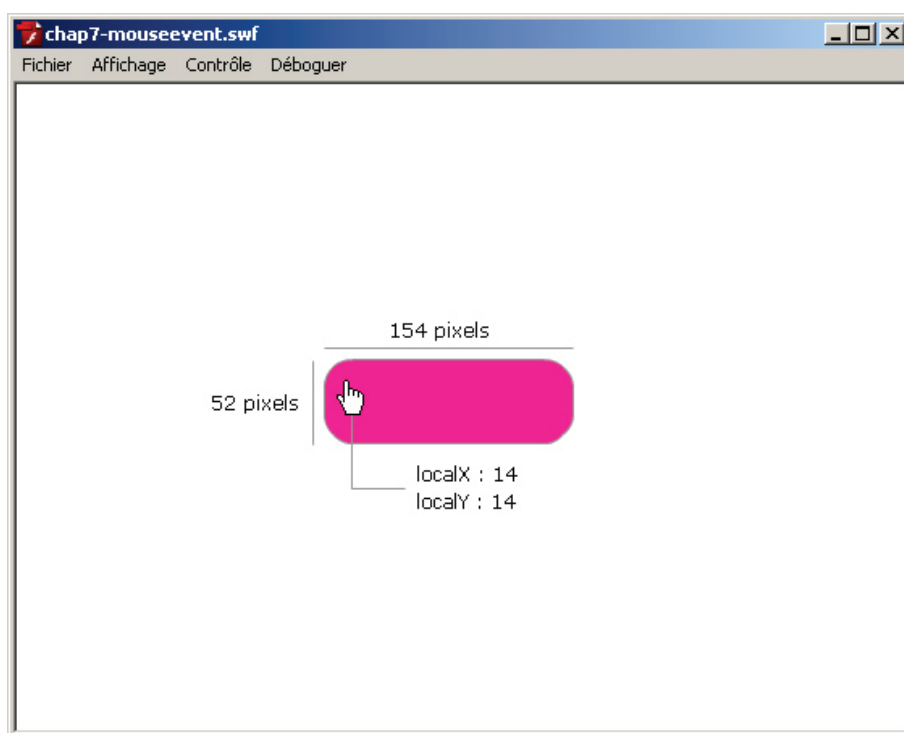
```
| }
```

Ces propriétés nous permettent par exemple de savoir à quelle position se trouve la souris par rapport aux coordonnées de l'objet survolé :

```
// souscription auprès de l'événement MouseEvent du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

function bougeSouris ( pEvt:MouseEvent ):void
{
    /*affiche :
    Position X de la souris par rapport au bouton : 14
    Position Y de la souris par rapport au bouton : 14
    */
    trace("Position X de la souris par rapport au bouton : " + pEvt.localX);
    trace("Position Y de la souris par rapport au bouton : " + pEvt.localY);
}
```

La figure 7-14 illustre l'intérêt des propriétés `localX` et `localY` :



*Figure 7-14. Propriétés `localX` et `localY`.*

Afin de savoir où se trouve la souris par rapport au `stage` et donc au scénario principal nous utilisons toujours les propriétés `stageX` et `stageY` de l'objet événementiel de type `MouseEvent` :

```
// souscription auprès de l'événement MouseEvent du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

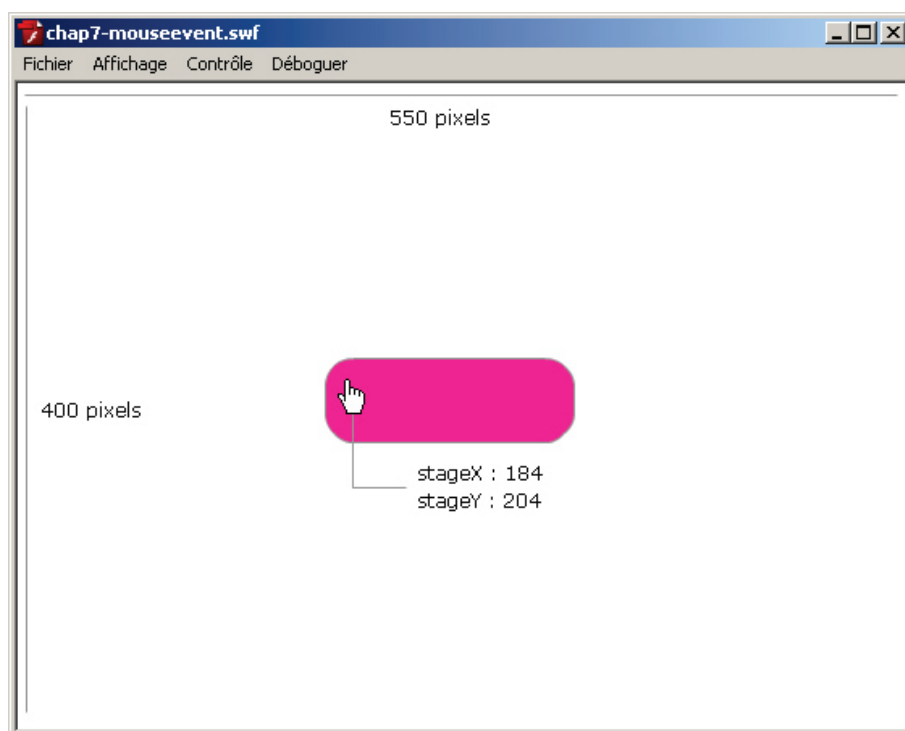
function bougeSouris ( pEvt:MouseEvent ):void
```

```

{
    /*affiche :
    Position X par rapport à la scène : 184
    Position Y par rapport à la scène : 204
    */
    trace("Position X de la souris par rapport à la scène : " + pEvt.stageX);
    trace("Position Y de la souris par rapport à la scène : " + pEvt.stageY);
}

```

La figure 7-15 illustre les propriétés `stageX` et `stageY` :



*Figure 7-15. Propriétés `stageX` et `stageY`.*

Notons que lorsque l'objet `Stage` est la cible d'un événement souris, les propriétés `stageX`, `stageY` et `localX` et `localY` de l'objet événementiel renvoient la même valeur.

## A retenir

- Les propriétés `localX` et `localY` renvoient les coordonnées de la souris par rapport aux coordonnées locale de l'objet cliqué.
- Les propriétés `stageX` et `stageY` renvoient les coordonnées de la souris par rapport au scénario principal.

## Événement global

En ActionScript 1 et 2 les clips étaient capables d'écouter tous les événements souris, même si ces derniers étaient invisibles ou non cliquables. Pour écouter les déplacements de la souris sur la scène nous pouvons écrire :

```
// définition d'un gestionnaire d'événement
monClip.onMouseMove = function ()
{
    // affiche : 78 : 211
    trace( _xmouse + " : " + _ymouse );
}
```

En définissant une fonction anonyme sur la propriété `onMouseMove` nous disposons d'un moyen efficace d'écouter les déplacements de la souris sur toute l'animation. En ActionScript 3 les comportements souris ont été modifiés, si nous écoutons des événements souris sur un objet non présent au sein de la liste d'affichage, l'événement n'est pas diffusé, car aucune interaction n'intervient entre la souris et l'objet graphique.

A l'inverse, si l'objet est visible l'événement n'est diffusé que lorsque la souris se déplace au-dessus de l'objet. Pour nous en rendre compte, nous posons une occurrence de symbole bouton sur le scénario principal que nous appelons `monBouton` et nous écoutons l'événement `MouseEvent.MOUSE_MOVE` :

```
// souscription auprès de l'événement MouseEvent du bouton
monBouton.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

function bougeSouris ( pEvt:MouseEvent ):void
{
    trace("Déclenché uniquement lors du déplacement de la souris sur le bouton
monBouton");
}
```

Afin de mettre en application la notion d'événements globaux, nous allons développer une application de dessin, nous ajouterons plus tard de nouvelles fonctionnalités à celle-ci comme l'export JPEG et PNG afin de sauvegarder notre dessin.

## Mise en application

Pour mettre en application la notion d'événement global nous allons créer ensemble une application de dessin dans laquelle l'utilisateur dessine à l'écran à l'aide de la souris. La première étape pour ce type d'application consiste à utiliser les capacités de dessin offertes par la classe propriété `graphics` de tout objet de type `flash.display.DisplayObject`.

Notons que l'API de dessin n'est plus directement disponible sur la classe `MovieClip` mais depuis la propriété `graphics` de tout `DisplayObject`.

Dans un nouveau document Flash CS3 nous allons tout d'abord nous familiariser avec l'API de dessin. Contrairement à l'API disponible en ActionScript 1 et 2 qui s'avérait limitée, la version ActionScript 3 a été enrichie. Afin de dessiner nous allons créer un `DisplayObject` le plus simple possible et le plus optimisé, pour cela nous nous orientons vers la classe `flash.display.Shape` :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

// ajout à la liste d'affichage
addChild ( monDessin );
```

Nous allons reproduire le même mécanisme que dans la réalité en traçant à partir du point cliqué. Le premier événement à écouter est donc l'événement `MouseEvent.CLICK`, pour détecter le premier clic et déplacer la mine à cette position :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

// ajout à la liste d'affichage
addChild ( monDessin );

// souscription auprès de l'événement MouseEvent.CLICK du scénario
stage.addEventListener ( MouseEvent.CLICK, clicSouris );

function clicSouris ( pEvt:MouseEvent ):void
{
    // position de la souris
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;
}
```

A chaque clic souris nous récupérons la position de la souris, nous allons à présent déplacer la mine en appelant la méthode `moveTo` :

```
// création du conteneur de tracés vectoriels
```

```
var monDessin:Shape = new Shape();

// ajout à la liste d'affichage
addChild ( monDessin );

// souscription auprès de l'événement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

function clicSouris ( pEvt:MouseEvent ):void
{
    // position de la souris
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monDessin.graphics.moveTo ( positionX, positionY );
}
```

Pour l'instant rien n'est dessiné lorsque nous cliquons car nous ne faisons que déplacer la mine de notre stylo imaginaire mais nous ne lui ordonnons pas de tracer, afin de dessiner nous devons appeler la méthode `lineTo` en permanence lorsque notre souris est en mouvement. Il nous faut donc écouter un nouvel événement.

La première chose à ajouter est l'écoute de l'événement `MouseEvent.MOUSE_MOVE` qui va nous permettre de déclencher une fonction qui dessinera lorsque la souris sera déplacée :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

// ajout à la liste d'affichage
addChild ( monDessin );

// souscription auprès de l'événement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

function clicSouris ( pEvt:MouseEvent ):void
{
    // position de la souris
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monDessin.graphics.moveTo ( positionX, positionY );

    // lorsque la souris est cliquée nous commençons l'écoute de celle ci
    pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

function bougeSouris ( pEvt:MouseEvent ):void
```

```

{

    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monDessin.graphics.lineTo ( positionX, positionY );

}

```

Si nous testons le code précédent, aucun tracé n'apparaîtra car nous n'avons pas défini le style du tracé avec la méthode `lineStyle` de la classe `Graphics` :

```

// initialisation du style de tracé
monDessin.graphics.lineStyle ( 1, 0x990000, 1 );

```

Afin de ne pas continuer à tracer lorsque l'utilisateur relâche la souris nous devons supprimer l'écoute de l'événement

`MouseEvent.MOUSE_MOVE` lorsque l'événement

`MouseEvent.MOUSE_UP` est diffusé :

```

// souscription auprès de l'événement MouseEvent.MOUSE_UP du scénario
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function relacheSouris ( pEvt:MouseEvent ):void

{

    // désinscription auprès de l'événement MouseEvent.MOUSE_MOVE
    pEvt.currentTarget.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

```

Tout fonctionne très bien, mais en regardant attentivement nous voyons que le rendu du tracé n'est pas très fluide. Si nous augmentons la cadence de l'animation à environ 60 img/sec nous remarquons que le rendu est plus fluide.

Il existe néanmoins une méthode beaucoup plus optimisée pour améliorer le rafraîchissement du rendu, nous allons intégrer cette fonctionnalité dans notre application.

## A retenir

- Seul l'objet `Stage` permet d'écouter la souris de manière globale.

## Mise à jour du rendu

C'est à l'époque du lecteur Flash 5 que la fonction `updateAfterEvent` a vu le jour. Cette fonction permet de mettre à jour le rendu du lecteur indépendamment de la cadence de l'animation. Dès que la fonction `updateAfterEvent` est déclenchée

le lecteur rend à nouveau les vecteurs, en résumé cette fonction permet de mettre à jour l’affichage entre deux images clés.

Cette fonction n’a d’intérêt qu’au sein de fonctions n’étant pas liées à la cadence de l’animation. C’est pour cette raison qu’en ActionScript 3, seules les classes `MouseEvent`, `KeyboardEvent` et `TimerEvent` possèdent une méthode `updateAfterEvent`.

Afin d’optimiser le rendu des tracés dans notre application de dessin nous forçons le rafraîchissement du rendu au sein de la fonction `bougeSouris` :

```
function bougeSouris ( pEvt:MouseEvent ):void
{
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monDessin.graphics.lineTo ( positionX, positionY );

    // force le rendu
    pEvt.updateAfterEvent();
}
```

Voici le code final de notre application de dessin :

```
// création du conteneur de tracés vectoriels
var monDessin:Shape = new Shape();

// ajout à la liste d'affichage
addChild ( monDessin );

// initialisation du style de tracé
monDessin.graphics.lineStyle ( 1, 0x990000, 1 );

// souscription auprès de l'événement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

// souscription auprès de l'événement MouseEvent.MOUSE_UP du scénario
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function clicSouris ( pEvt:MouseEvent ):void
{
    // position de la souris
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monDessin.graphics.moveTo ( positionX, positionY );

    // lorsque la souris est cliquée nous commençons l'écoute de celle ci
```



```
pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );  
}  
  
function bougeSouris ( pEvt:MouseEvent ):void  
{  
  
    var positionX:Number = pEvt.stageX;  
    var positionY:Number = pEvt.stageY;  
  
    // la mine est déplacée à cette position  
    // pour commencer à dessiner à partir de cette position  
    monDessin.graphics.lineTo ( positionX, positionY );  
  
    // force le rendu  
    pEvt.updateAfterEvent();  
  
}  
  
function relacheSouris ( pEvt:MouseEvent ):void  
{  
  
    // désinscription auprès de l'évènement MouseEvent.MOUSE_MOVE  
    pEvt.currentTarget.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );  
}  
  
}
```

Sans augmenter la cadence de notre animation, nous améliorons le rendu des tracés, en conservant une cadence réduite notre application consomme peu de ressources sans mettre de côté les performances.

Grâce à la propriété `frameRate` de la classe `Stage` nous réduisons la cadence de l'animation à l'exécution :

```
stage.frameRate = 2;
```

Avec une cadence réduite à deux image/sec nos tracés restent fluides. Intéressons-nous désormais à une autre notion liée à l'interactivité, la gestion du clavier.

## A retenir

- Grâce à la méthode `updateAfterEvent`, le rendu peut être forcé entre deux images. Il en résulte d'un meilleur rafraichissement de l'affichage.

## Gestion du clavier

Grâce au clavier nous pouvons ajouter une autre dimension à nos applications, cette fonctionnalité n'est d'ailleurs aujourd'hui pas assez utilisée dans les sites traditionnels, nous allons capturer certaines

touches du clavier afin d'intégrer de nouvelles fonctionnalités dans notre application de dessin créée auparavant.

Il serait judicieux d'intégrer un raccourci clavier afin d'effacer le dessin en cours. Par défaut, l'objet `Stage` gère les entrées clavier. Deux événements sont liés à l'écoute du clavier :

- `KeyboardEvent.KEY_DOWN` : diffusé lorsqu'une touche du clavier est enfoncée.
- `KeyboardEvent.KEY_UP` : diffusé lorsqu'une touche du clavier est relâchée.

Nous allons écouter l'événement `KeyboardEvent.KEY_DOWN` auprès de l'objet `Stage`, à chaque fois qu'une touche est enfoncée la fonction écouteur `ecouteClavier` est déclenchée :

```
// souscription auprès de l'objet Stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{
    // affiche : [KeyboardEvent type="keyDown" bubbles=true cancelable=false
    // eventPhase=2 charCode=114 keyCode=82 keyLocation=0 ctrlKey=false altKey=false
    // shiftKey=false]
    trace( pEvt );
}
```

Un objet événementiel de type `KeyboardEvent` est diffusé, cette classe possède de nombreuses propriétés dont voici le détail :

- `KeyboardEvent.altKey` : indique si la touche ALT est enfoncée, cette propriété n'est pas prise en charge pour le moment.
- `KeyboardEvent.charCode` : contient le code caractère Unicode correspondant à la touche du clavier.
- `KeyboardEvent.ctrlKey` : indique si la touche CTRL du clavier est enfoncée.
- `KeyboardEvent.keyCode` : valeur de code correspondant à la touche enfoncée ou relâchée.
- `KeyboardEvent.keyLocation` : indique l'emplacement de la touche sur le clavier.
- `KeyboardEvent.shiftKey` : indique si la touche SHIFT du clavier est enfoncée.

## Déterminer la touche appuyée

La propriété `charCode` de l'objet événementiel diffusé lors d'un événement clavier nous permet de déterminer la touche enfoncée.

Grâce à la méthode `String.fromCharCode` nous évaluons le code Unicode et récupérons le caractère correspondant :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

    // affiche : d,f,g, etc...
    trace( String.fromCharCode( pEvt.charCode ) );

}
```

Pour tester la touche appuyée, nous utilisons les propriétés statiques de la classe `Keyboard`. Les codes de touche les plus courants sont stockés au sein de propriétés statiques de la classe `Keyboard`.

Pour savoir si la touche espace est enfoncée nous écrivons :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

    if ( pEvt.keyCode == Keyboard.SPACE )

    {

        trace("Touche ESPACE enfoncée");

    }

}
```

Nous allons supprimer le dessin lorsque la touche ESPACE sera enfoncée, afin de pouvoir commencer un nouveau dessin :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{

    if ( pEvt.keyCode == Keyboard.SPACE )

    {

        // non effaçons tous les précédent tracés
        monDessin.graphics.clear();

        // Attention, nous devons obligatoirement redéfinir un style
        monDessin.graphics.lineStyle ( 1, 0x990000, 1 );

    }

}
```

Nous pourrions changer la couleur des tracés pour chaque nouveau dessin :

```
// souscription auprès de l'objet stage pour l'événement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{
    if ( pEvt.keyCode == Keyboard.SPACE )
    {
        // non effaçons tout les précédent tracés
        monDessin.graphics.clear();

        // Attention, nous devons obligatoirement redéfinir un style
        monDessin.graphics.lineStyle ( 1, Math.random()*0xFFFFFF, 1 );
    }
}
```

Nous reviendrons sur la manipulation avancée des couleurs au cours du chapitre 12 intitulé *Programmation bitmap*.

## Gestion de touches simultanées

En ActionScript 1 et 2 la gestion des touches simultanées avec le clavier n'était pas facile. En ActionScript 3 grâce aux propriétés de la classe `MouseEvent` nous pouvons très facilement détecter la combinaison de touches. Celle-ci est rendue possible grâce aux propriétés `altKey`, `ctrlKey` et `shiftKey`.

Nous allons ajouter la notion d'historique dans notre application de dessin. Lorsque l'utilisateur cliquera sur CTRL+Z nous reviendrons en arrière, pour repartir en avant l'utilisateur cliquera sur CTRL+Y.

Pour concevoir cette fonctionnalité nous avons besoin de pouvoir dissocier chaque tracé, pour cela nous allons intégrer chaque tracé dans un objet `Shape` différent et l'ajouter à un conteneur principal. Pour pouvoir contenir des objets `Shape`, notre conteneur principal devra être un `DisplayObjectContainer`, nous modifions donc les premières lignes de notre application pour intégrer un conteneur de type `Sprite` :

```
// création du conteneur de tracés vectoriels
var monDessin:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( monDessin );
```

Nous devons stocker au sein de deux tableaux les formes supprimées et les formes affichées :

```
// tableau référençant les formes tracées et supprimées
var tableauTraces:Array = new Array();
var tableauAncienTraces:Array = new Array();
```

A chaque fois que la souris sera cliquée, nous devons créer un objet **Shape** spécifique à chaque tracé, ce dernier est alors référencé au sein du tableau **tableauTraces**. Nous modifions le corps de la fonction **clicSouris** :

```
function clicSouris ( pEvt:MouseEvent ):void
{
    // position de la souris
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // un nouvel objet Shape est créé pour chaque tracé
    var monNouveauTrace:Shape = new Shape();

    // nous ajoutons le conteneur de tracé au conteneur principal
    monDessin.addChild ( monNouveauTrace );

    // puis nous référençons le tracé au sein du tableau
    // référençant les tracés affichés
    tableauTraces.push ( monNouveauTrace );

    // nous définissons un style de tracé
    monNouveauTrace.graphics.lineStyle ( 1, 0x990000, 1 );

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monNouveauTrace.graphics.moveTo ( positionX, positionY );

    // si un nouveau tracé intervient alors que nous sommes
    // repartis en arrière nous repartons de cet état
    if ( tableauAncienTraces.length ) tableauAncienTraces = new Array();

    // lorsque la souris est cliquée nous commençons l'écoute de celle ci
    pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}
```

Lorsque la souris est en mouvement nous devons tracer au sein du dernier objet **Shape** ajouté, grâce à notre tableau **tableauTraces** nous récupérons le dernier objet **Shape** et traçons au sein de ce dernier :

```
function bougeSouris ( pEvt:MouseEvent ):void
{
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // nous récupérons le dernier objet Shape ajouté
    // prêt à accueillir le nouveau tracé
    var monNouveauTrace:Shape = tableauTraces[tableauTraces.length-1];

    // la mine est déplacée à cette position
```

```
// pour commencer à dessiner à partir de cette position
monNouveauTrace.graphics.lineTo ( positionX, positionY );

// force le rafraichissement du rendu
pEvt.updateAfterEvent();

}
```

La dernière fonction que nous devons modifier est la fonction `ecouteClavier`, c'est au sein de celle-ci que nous testons la combinaison de touches et décidons s'il faut supprimer des tracés ou les réafficher :

```
function ecouteClavier ( pEvt:KeyboardEvent ):void
{
    // si la barre espace est enfoncée
    if ( pEvt.keyCode == Keyboard.SPACE )
    {
        // nombre d'objets Shape contenant des tracés
        var lng:int = tableauTraces.length;

        // suppression des tracés de la liste d'affichage
        while ( lng-- ) monDessin.removeChild ( tableauTraces[lng] );

        // les tableaux d'historiques sont reinitialisés
        // les références supprimées
        tableauTraces = new Array();
        tableauAncienTraces = new Array();
    }

    // code des touches Z et Y
    var ZcodeTouche:int = 90;
    var YcodeTouche:int = 89;

    if ( pEvt.ctrlKey )
    {
        // si retour en arrière (CTRL+Z)
        if( pEvt.keyCode == ZcodeTouche && tableauTraces.length )
        {
            // nous supprimons le dernier tracé
            var aSupprimer:Shape = tableauTraces.pop()

            // nous stockons chaque tracé supprimé dans le tableau spécifique
            tableauAncienTraces.push ( aSupprimer );

            // nous supprimons le tracé de la liste d'affichage
            monDessin.removeChild( aSupprimer );

            // si retour en avant (CTRL+Y)
        } else if ( pEvt.keyCode == YcodeTouche &&
tableauAncienTraces.length )
        {

```

```
        // nous récupérons le dernier tracé ajouté
        var aAfficher:Shape = tableauAncienTraces.pop();

        // nous le remplaçons dans le tableau de tracés à l'affichage
        tableauTraces.push ( aAfficher );

        // puis nous l'affichons
        monDessin.addChild ( aAfficher );

    }

}

}
```

Voici le code complet de notre application de dessin avec gestion de l'historique :

```
// création du conteneur de tracés vectoriels
var monDessin:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( monDessin );

// souscription auprès de l'évènement MouseEvent.MOUSE_DOWN du scénario
stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

// souscription auprès de l'évènement MouseEvent.MOUSE_UP du scénario
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

// tableaux référençant les formes tracées et supprimées
var tableauTraces:Array = new Array();
var tableauAncienTraces:Array = new Array();

function clicSouris ( pEvt:MouseEvent ):void
{

    // position de la souris
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // un nouvel objet Shape est créé pour chaque tracé
    var monNouveauTrace:Shape = new Shape();

    // nous ajoutons le conteneur de tracé au conteneur principal
    monDessin.addChild ( monNouveauTrace );

    // puis nous référençons le tracé au sein du tableau
    // référençant les tracés affichés
    tableauTraces.push ( monNouveauTrace );

    // nous définissons un style de tracé
    monNouveauTrace.graphics.lineStyle ( 1, 0x990000, 1 );

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monNouveauTrace.graphics.moveTo ( positionX, positionY );

    // si un nouveau tracé intervient alors que nous sommes repartis
    // en arrière nous repartons de cet état
    if ( tableauAncienTraces.length ) tableauAncienTraces = new Array;
```

```
// lorsque la souris est cliquée nous commençons l'écoute de celle ci
pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

function bougeSouris ( pEvt:MouseEvent ):void
{
    var positionX:Number = pEvt.stageX;
    var positionY:Number = pEvt.stageY;

    // nous récupérons le dernier objet Shape ajouté
    // prêt à accueillir le nouveau tracé
    var monNouveauTrace:Shape = tableauTraces[tableauTraces.length-1];

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monNouveauTrace.graphics.lineTo ( positionX, positionY );

    // force le rafraichissement du rendu
    pEvt.updateAfterEvent();
}

function relacheSouris ( pEvt:MouseEvent ):void
{
    // désinscription auprès de l'évènement MOUSE_MOVE
    pEvt.currentTarget.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

// souscription auprès de l'objet stage pour l'évènement KEY_DOWN
stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

function ecouteClavier ( pEvt:KeyboardEvent ):void
{
    // si la barre espace est enfoncée
    if ( pEvt.keyCode == Keyboard.SPACE )

    {

        // nombre d'objets Shape contenant des tracés
        var lng:int = tableauTraces.length;

        // suppression des tracés de la liste d'affichage
        while ( lng-- ) monDessin.removeChild ( tableauTraces[lng] );

        // les tableaux d'historiques sont reinitialisés
        // les références supprimées
        tableauTraces = new Array();
        tableauAncienTraces = new Array();

    }

    // code des touches Z et Y
    var ZcodeTouche:int = 90;
```



```
var YcodeTouche:int = 89;

if ( pEvt.ctrlKey )
{
    // si retour en arrière (CTRL+Z)
    if( pEvt.keyCode == ZcodeTouche && tableauTraces.length )
    {
        // nous supprimons le dernier tracé
        var aSupprimer:Shape = tableauTraces.pop()

        // nous stockons chaque tracé supprimé dans le tableau spécifique
        tableauAncienTraces.push ( aSupprimer );

        // nous supprimons le tracé de la liste d'affichage
        monDessin.removeChild( aSupprimer );

        // si retour en avant (CTRL+Y)
        } else if ( pEvt.keyCode == YcodeTouche &&
        tableauAncienTraces.length )
        {
            // nous récupérons le dernier tracé ajouté
            var aAfficher:Shape = tableauAncienTraces.pop();

            // nous le remplaçons dans le tableau de tracés à l'affichage
            tableauTraces.push ( aAfficher );

            // puis nous l'affichons
            monDessin.addChild ( aAfficher );
        }
    }
}
```

Nous verrons au cours du chapitre 10 intitulé *Programmation Bitmap* comment capturer des données vectorielles afin de les compresser et les exporter en un format d'image spécifique type PNG ou JPEG.

---

Attention : Lors du test de cette application, veuillez à bien cocher l'option *Désactiver les raccourcis clavier* du menu *Contrôle* du lecteur Flash. Autrement les touches correspondant aux outils de l'environnement auteur de Flash ne pourront être détectées par le lecteur.

---

Nous pourrions imaginer une application permettant à l'utilisateur de dessiner, puis de sauvegarder sur son ordinateur le dessin au format favori. ActionScript nous réserve encore bien des surprises !

---

## A retenir

---

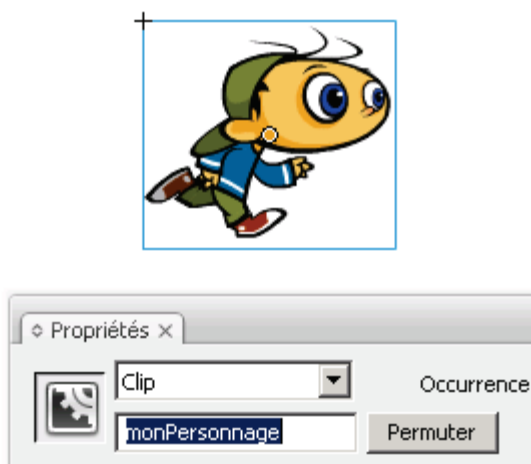
- La classe `MouseEvent` permet grâce aux propriétés `shiftKey` et `ctrlKey` la pression de touches simultanées.

## Simuler la méthode `Key.isDown`

ActionScript 3 n'intègre pas de méthode équivalente à la méthode `isDown` de la classe `Key` existante en ActionScript 1 et 2.

Dans le cas de développements de jeux il peut être important de simuler cet ancien comportement. Nous allons nous y attarder à présent. Afin de développer cette fonctionnalité, nous utilisons un symbole de type clip afin de le déplacer sur la scène à l'aide du clavier.

La figure 7-16 illustre le symbole :



*Figure 7-16. Occurrence de symbole clip.*

Grâce à un objet de mémorisation, nous pouvons reproduire le même comportement que la méthode `isDown` à l'aide du code suivant :

```
// écoute des événements clavier
stage.addEventListener ( KeyboardEvent.KEY_DOWN, toucheEnfoncée );
stage.addEventListener ( KeyboardEvent.KEY_UP, toucheRelachée );

// objet de mémorisation de l'état des touches
var touches:Object = new Object();

function toucheEnfoncée ( pEvt:KeyboardEvent ):void
{
    // marque la touche en cours comme enfoncée
    touches [ pEvt.keyCode ] = true;
}
```

```
function toucheRelachee ( pEvt:KeyboardEvent ):void
{
    // marque la touche en cours comme relachée
    touches [ pEvt.keyCode ] = false;
}

monPersonnage.addEventListener ( Event.ENTER_FRAME, deplace );

var etat:Number = personnage_mc.scaleX;
var vitesse:Number = 15;

function deplace ( pEvt:Event ):void
{
    // si la touche Keyboard.RIGHT est enfoncée
    if ( touches [ Keyboard.RIGHT ] )
    {
        pEvt.target.x += vitesse;
        pEvt.target.scaleX = etat;

        // si la touche Keyboard.LEFT est enfoncée
    } else if ( touches [ Keyboard.LEFT ] )
    {
        pEvt.target.x -= vitesse;
        pEvt.target.scaleX = -etat;
    }
}
```

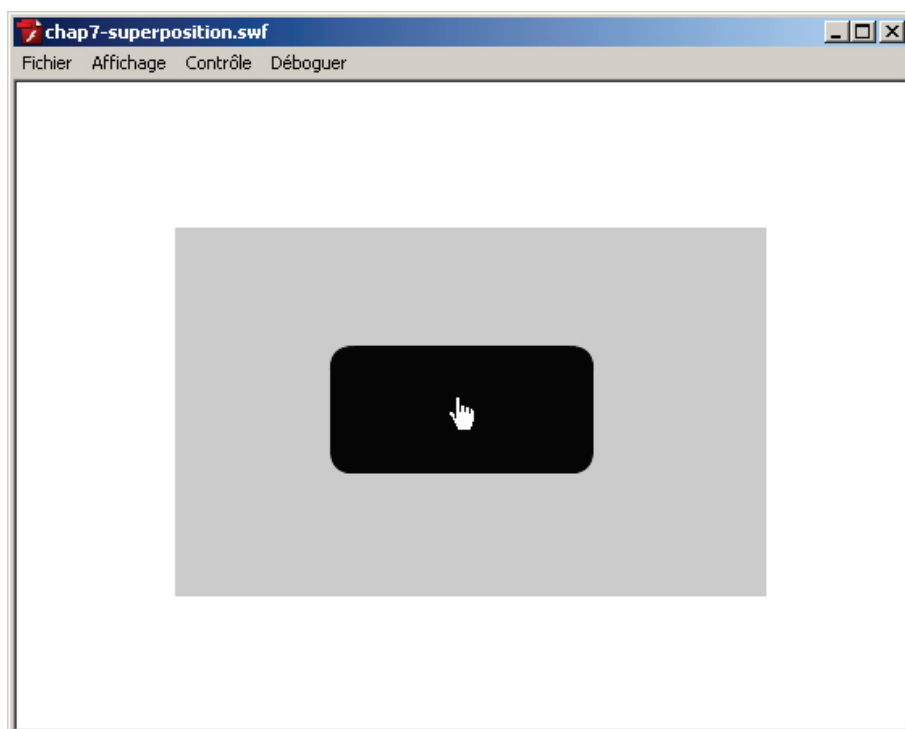
En testant le code précédent, le symbole est manipulé par le clavier. Nous venons de simuler le fonctionnement de la méthode `Key.isDown` qui n'existe plus en ActionScript 3.

## A retenir

- La propriété `keyCode` de l'objet événementiel de type `KeyboardEvent` renvoie le code de la touche.
- Il n'existe pas d'équivalent à la méthode `isDown` de la classe `Key` en ActionScript 3. Il est en revanche facile de simuler un comportement équivalent.

## Superposition

Le lecteur 9 en ActionScript 3 intègre un nouveau comportement concernant la superposition des objets graphiques. En ActionScript 1 et 2 lorsqu'un objet non cliquable était superposé sur un bouton, le bouton recouvert recevait toujours les entrées souris.

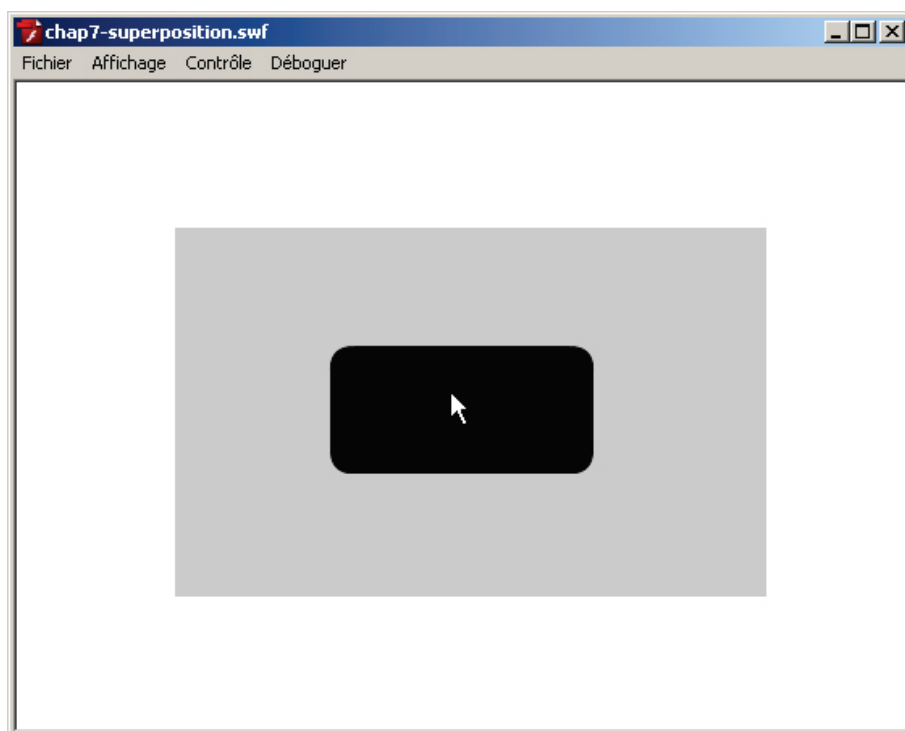


*Figure 7-17. Le bouton reste cliquable même recouvert.*

La figure 7-17 illustre une animation ActionScript 1/2, un clip de forme rectangulaire à opacité réduite recouvre un bouton qui demeure cliquable et affiche le curseur représentant une main. Si nous cliquons sur le bouton, son événement `onRelease` était déclenché.

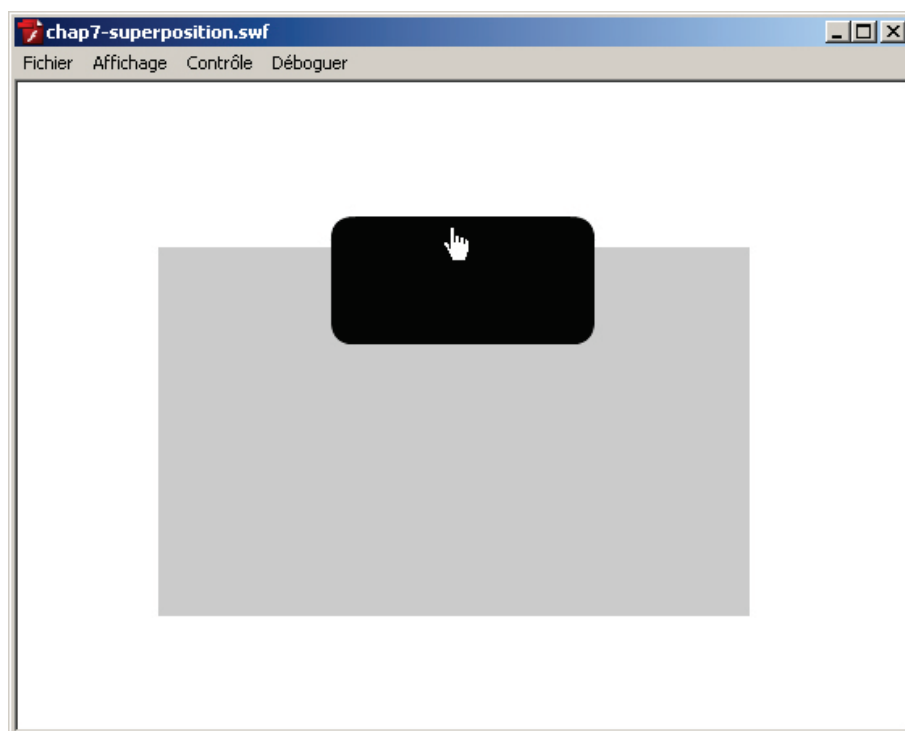
En ActionScript 3 le lecteur 9 prend l'objet le plus haut de la hiérarchie, les objets étant censés recevoir des entrées souris recouverts par un objet graphique même non cliquable ne reçoivent plus les entrées les souris et n'affichent pas le curseur main.

La figure 7-18 illustre le comportement en ActionScript 3 :



*Figure 7-18. Le bouton n'est plus cliquable si recouvert.*

Si l'objet placé au-dessus découvre une partie du bouton, cette partie devient alors cliquable :



*Figure 7-19. Zone cliquable sur bouton.*

Afin de rendre un objet superposé par un autre cliquable nous devons passer par la propriété `mouseEnabled` définie par la classe `InteractiveObject`. En passant la valeur `false` à la propriété `mouseEnabled` de l'objet superposé nous rendant les objets placés en dessous sensibles aux entrées souris.

En ayant au préalable nommé le clip superposé `monClip`, nous rendons cliquable le bouton situé en dessous :

```
// désactive la réaction aux entrées souris
monClip.mouseEnabled = false;
```

Nous pouvons en déduire que lorsqu'une occurrence de symbole recouvre un objet cliquable, ce dernier ne peut recevoir d'événement souris. A l'inverse lorsqu'une occurrence de symbole contient un objet cliquable, ce dernier continue de recevoir des événements souris. C'est un comportement que nous avons traité en début de chapitre lors de la construction du menu. Les objets enfants des occurrences de `Sprite` continuaient de recevoir les entrées souris et rendaient nos boutons partiellement cliquables.

Afin de désactiver tous les objets enfants, nous avons utilisé la propriété `mouseChildren`, qui revient à passer la propriété `mouseEnabled` de tous les objets enfants d'un `DisplayObjectContainer`.

## L'événement `Event.RESIZE`

Lorsque l'animation est redimensionnée un événement `Event.RESIZE` est diffusé par l'objet `Stage`. Grace à cet événement nous pouvons gérer le redimensionnement de notre animation. Dans un nouveau document Flash CS3 nous créons un clip contenant le logo de notre site et nommons l'occurrence `monLogo`.

Le code suivant nous permet de conserver notre logo toujours centré quel que soit le redimensionnement effectué sur notre animation :

```
// import des classes de mouvement
import fl.transitions.Tween;
import fl.transitions.easing.Strong;

// alignement de la scène en haut à gauche
stage.align = StageAlign.TOP_LEFT;
// nous empêchons le contenu d'être étiré
stage.scaleMode = StageScaleMode.NO_SCALE;

// écoute de l'événement auprès de l'objet Stage
stage.addEventListener ( Event.RESIZE, redimensionne );

// calcul de la position en x et y
```

```

var initCentreX:int = (stage.stageWidth - monLogo.width)/2;
var initCentreY:int = (stage.stageHeight - monLogo.height)/2;

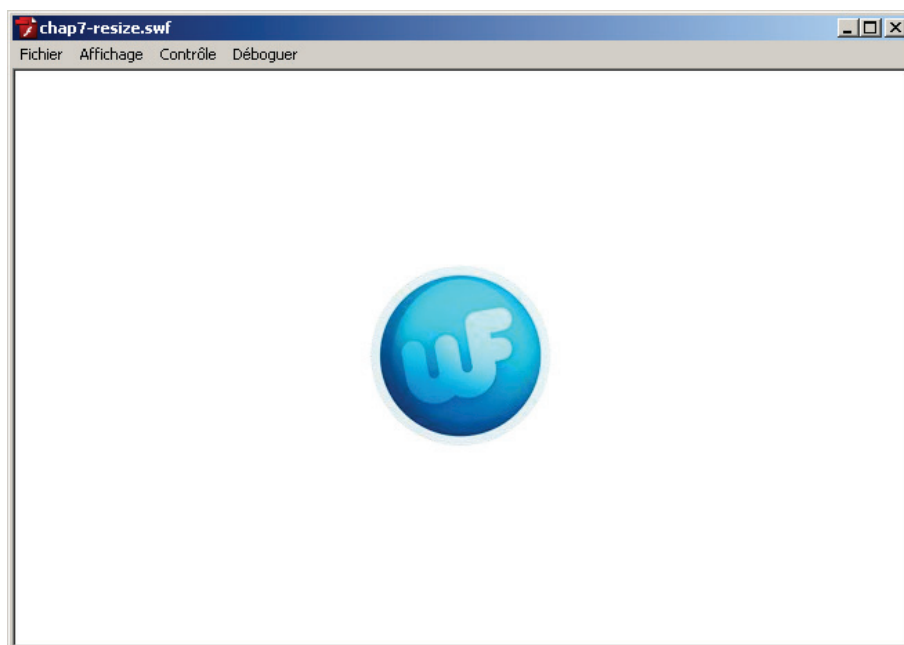
// deux objets Tween sont créés pour chaque axe
var tweenX:Tween = new Tween ( monLogo, "x", Strong.easeOut, monLogo.x,
initCentreX, 15 );
var tweenY:Tween = new Tween ( monLogo, "y", Strong.easeOut, monLogo.y,
initCentreY, 15 );

function redimensionne ( pEvt : Event ):void
{
    // calcul de la position en x et y
    var centreX:int = (stage.stageWidth - monLogo.width)/2;
    var centreY:int = (stage.stageHeight - monLogo.height)/2;

    // nous affectons les valeurs de départ et d'arrivée et relançons le
    mouvement
    tweenX.begin = monLogo.x;
    tweenX.finish = centreX;
    tweenX.start();
    tweenY.begin = monLogo.y;
    tweenY.finish = centreY;
    tweenY.start();
}

```

La figure 7-20 illustre le résultat :



*Figure 7-20. Logo centré automatiquement.*

Nous pouvons utiliser l'événement `Event.RESIZE` afin de gérer le repositionnement d'un ensemble d'éléments graphiques au sein d'une application ou d'un site.

Les précédents chapitres nous ont permis de comprendre certains mécanismes avancés d'ActionScript 3 que nous avons traités de manière procédurale.

Nous allons durant les prochains chapitres concevoir nos applications à l'aide d'objets communiquant, ce style de programmation est appelé programmation orientée objet ou plus communément POO.



# 8

## Programmation orientée objet

<b>CONCEVOIR AUTREMENT .....</b>	<b>1</b>
TOUT EST OBJET .....	4
<b>NOTRE PREMIERE CLASSE.....</b>	<b>9</b>
INTRODUCTION AUX PAQUETAGES .....	10
DÉFINITION DE PROPRIÉTÉS .....	12
ATTRIBUTS DE PROPRIÉTÉS DE CLASSE .....	13
ATTRIBUTS DE CLASSE .....	15
LE CONSTRUCTEUR .....	16
UTILISATION DU MOT CLÉ THIS .....	21
DÉFINITION DE MÉTHODES .....	22
<b>L'ENCAPSULATION .....</b>	<b>25</b>
MISE EN PRATIQUE DE L'ENCAPSULATION .....	28
LES MÉTHODES D'ACCÈS.....	28
CONTRÔLE D'AFFECTATION .....	34
MÉTHODES EN LECTURE/ÉCRITURE.....	39
<b>CAS D'UTILISATION DE L'ATTRIBUT STATIC .....</b>	<b>45</b>
<b>LA CLASSE JOUEURMANAGER .....</b>	<b>51</b>
<b>L'HERITAGE .....</b>	<b>57</b>
SOUS-TYPES ET SUPER-TYPE .....	60
SPÉCIALISER UNE CLASSE .....	63
<b>LE TRANSTYPAGE .....</b>	<b>65</b>
<b>SURCHARGE .....</b>	<b>70</b>

### Concevoir autrement

La programmation orientée objet doit être considérée comme une manière de penser et de concevoir une application, certains la considère d'ailleurs comme un *paradigme de programmation*. Cette

nouvelle manière de penser n'est pas limitée à ActionScript 3 et peut être appliquée à un grand nombre de langages. Avant de démarrer, nous allons tout d'abord nous familiariser avec le vocabulaire puis nous entamerons notre aventure orientée objet à travers différents exemples concrets.

C'est en 1967 que le premier langage orienté objet vu le jour sous le nom de Simula-67. Comme son nom l'indique, Simula permettait de simuler toutes sortes de situations pour les applications de l'époque telles les applications de gestions, de comptabilité ou autres. Nous devons la notion d'objets, de classes, de sous-classes, d'événements ainsi que de ramasse-miettes (garbage collector) à Simula. Alan Key, inventeur du terme *programmation orientée objet* et ingénieur chez Xerox étendit les concepts apportés par Simula et développa le langage Smalltalk, aujourd'hui considéré comme le réel point de départ de la programmation orientée objet.

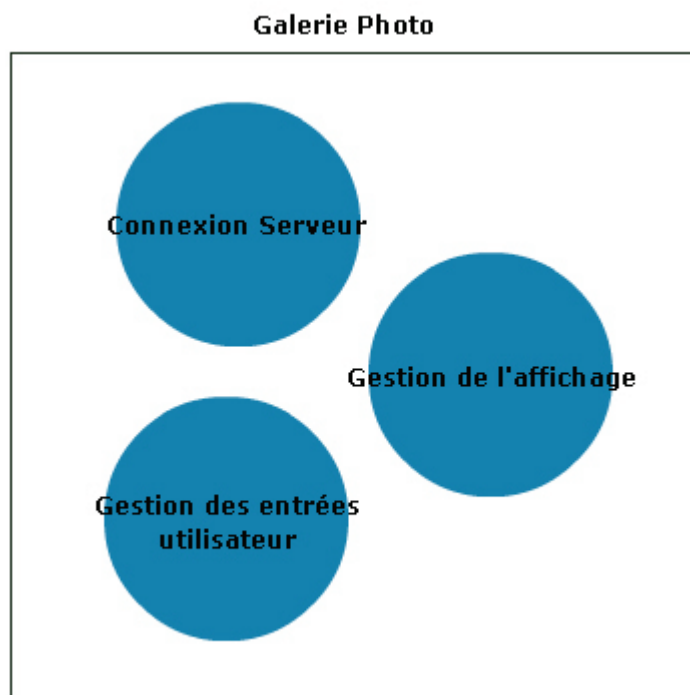
L'intérêt de la programmation orientée objet réside dans la *séparation des tâches*, *l'organisation* et la *réutilisation du code*. Nous pourrions résumer ce chapitre en une seule phrase essentielle à la base du concept de la programmation orientée objet : *A chaque type d'objet une tâche spécifique*.

Afin de développer un programme ActionScript nous avons le choix entre deux approches. La première, appelée couramment programmation procédurale ou fonctionnelle ne définit aucune séparation claire des tâches. Les différentes fonctionnalités de l'application ne sont pas affectées à un *objet spécifique*, des fonctions sont déclenchées dans un ordre précis et s'assurent que l'application s'exécute correctement.

En résumé, il s'agit de la première approche de programmation que tout développeur chevronné a connue. Au moindre bug ou mise à jour, c'est l'angoisse, aucun moyen d'isoler les comportements et les bogues, bienvenue dans le *code spaghetti*.

La seconde approche, appelée programmation orientée objet définit une séparation claire des tâches. Chaque objet s'occupe d'une tâche spécifique, dans le cas d'une galerie photo un objet peut gérer la connexion au serveur afin de récupérer les données, un autre s'occupe de l'affichage, enfin un dernier objet traite les entrées souris et clavier de l'utilisateur. En connectant ces objets entre eux, nous donnons vie à une application. Nous verrons plus tard que toute la difficulté de la programmation orientée objet, réside dans la communication inter objets.

La figure 8-1 illustre un exemple de galerie photo composée de trois objets principaux.



*Figure 8-1. Concept de séparation des tâches.*

Même si cela nous paraît naturel, la notion de programmation orientée objet reste abstraite sans un cas concret appliqué aux développements de tous les jours. Afin de simplifier tout cela, nous pouvons regarder les objets qui nous entourent. Nous retrouvons à peu près partout le concept de *séparation des tâches*.

Afin d’optimiser le temps de développement et de maintenance, un programme peut être considéré comme un ensemble d’objets communicants pouvant à tout moment être remplacés ou réutilisés. Une simple voiture en est l’exemple parfait.

Le moteur, les roues, l’embrayage, les pédales, permettent à notre voiture de fonctionner. Si celle-ci tombe en panne, chaque partie sera testée, au cas où l’une d’entre elles s’avère défectueuse, elle est aussitôt remplacée ou réparée permettant une réparation rapide de la voiture. En programmation, la situation reste la même, si notre application objet est boguée, grâce à la séparation des tâches nous pouvons très facilement isoler les erreurs et corriger l’application ou encore ajouter de nouvelles fonctionnalités en ajoutant de nouveaux objets.

## Tout est objet

Dans le monde qui nous entoure tout peut être représenté sous forme d'objet, une ampoule, une voiture, un arbre, la terre elle-même peut être considérée comme un objet.

En programmation, il faut considérer un objet comme une entité ayant un état et permettant d'effectuer des opérations. Pour simplifier nous pouvons dire qu'un objet est capable d'effectuer certaines tâches et possède des caractéristiques spécifiques.

Une télévision peut être considérée comme un objet, en possédant une taille, une couleur, ainsi que des fonctionnalités, comme le fait d'être allumée, éteinte ou en veille. Ici encore, la séparation des tâches est utilisée, le tube cathodique s'occupe de créer l'image, l'écran se charge de l'afficher, le capteur s'occupe de gérer les informations envoyées par la télécommande. Cet ensemble d'objets produit un résultat fonctionnel et utilisable. Nous verrons que de la même manière nous pouvons concevoir une application ActionScript 3 en associant chaque objet à une tâche spécifique.

Afin de porter notre exemple de télévision en ActionScript 3, nous pourrions définir une classe `Television` permettant de créer des objets télévisions. Chaque objet télévision serait donc de type `Television`. Lorsque un développeur parle de la classe `Array` nous savons de quel type d'objet il s'agit et de ses capacités, comme ajouter, supprimer ou trier des éléments. De la même manière, en parlant de la classe `Television` nous savons quelles sont les fonctionnalités offertes par ce type.

Si nous regardons à l'intérieur d'une télévision, nous y trouvons toutes sortes de composants. Des hauts parleurs, des boutons, ainsi qu'un ensemble de composants électroniques. En programmation, nous pouvons exactement reproduire cette conception, nous verrons que plusieurs objets travaillant ensemble peuvent donner naissance à une application concrète. Chaque composant peut ainsi être réutilisé dans un autre programme, les hauts parleurs d'un modèle de télévision pourront être réutilisés dans une autre.

C'est ce que nous appelons la notion de *composition*, nous reviendrons sur ce sujet au cours du chapitre 10 intitulé *Diffusion d'événements personnalisés*.

Une fois notre objet créé, nous pouvons lui demander d'exécuter certaines actions, lorsque nous achetons une télévision nous la choisissons selon ses capacités, et nous nous attendons au moins à pouvoir l'allumer, l'éteindre, monter le son ou baisser le son. Ces capacités propres à un objet sont définies par son *interface*, les

capacités de celle-ci sont directement liées au type. La figure 8-2 illustre l'exemple d'une télévision.

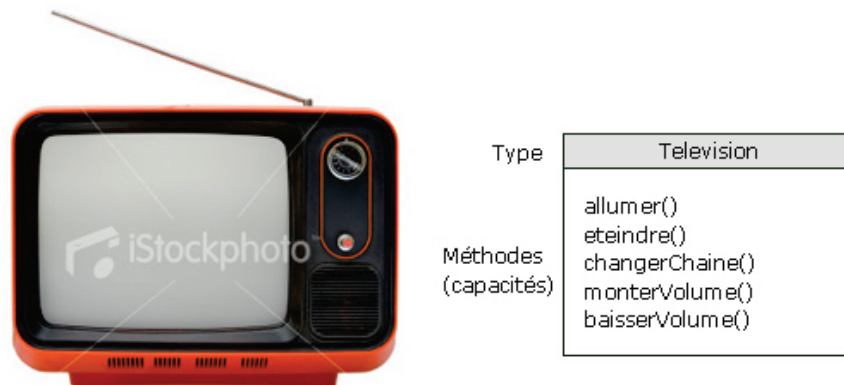


Figure 8-2. Classe *Television*.

Afin d'instancier un objet à partir d'une classe nous utilisons le mot clé `new` :

```
// création d'une instance de la classe Television au sein de la variable maTV
var maTV:Television = new Television();
```

Le type `Television` détermine l'interface de notre télévision, c'est-à-dire ce que l'objet est capable de faire. Les fonctions `allumer`, `eteindre`, `monterVolume`, et `baisserVolume` sont appelées méthodes car elles sont rattachées à un objet et définissent ses capacités. En les appelants, nous exécutons tout un ensemble de mécanismes totalement transparents pour l'utilisateur. Dans l'exemple suivant, nous instancions une télévision, s'il est trop tard nous l'éteignons :

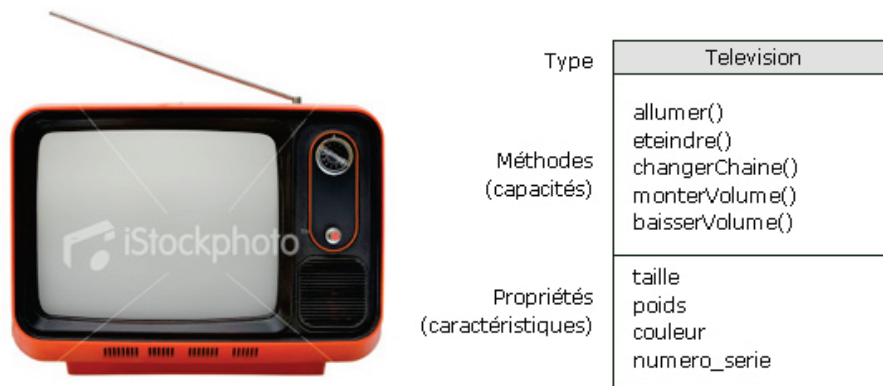
```
// création d'une instance de la classe Television au sein de la variable maTV
var maTV:Television = new Television();

// nous allumons la télé
maTV.allumer()

// si il est trop tard
if ( heureActuelle > 12 )
{
    // nous éteignons la télé
    maTV.eteindre();
}
```

Comment stockons-nous les informations telles la taille, la couleur, ou bien le numéro de série de la télévision ?

Bonne question, ces données représentent les caractéristiques d'un objet et sont stockées au sein de *propriétés*. Nous pouvons imaginer qu'une télévision ait une taille, un poids, une couleur, et un numéro de série.



*Figure 8-3. Méthodes et propriétés du type  
Television.*

Toutes les instances de télévisions créées à partir de la même classe possèdent ainsi les mêmes fonctionnalités, mais des caractéristiques différentes comme la couleur, la taille ou encore le poids. Pour résumer, nous pouvons dire que les méthodes et propriétés déterminent le type.

Pour récupérer la couleur ou le poids d'une télévision, nous ciblons la propriété voulue :

```
// nous récupérons la taille
var taille:Number = maTV.taille;

// nous récupérons la couleur
var couleur:Number = maTV.couleur;
```

Au sein de Flash nous retrouvons partout le concept d'objets, prenons le cas de la classe `MovieClip`, dans le code suivant nous créons une instance de `MovieClip` :

```
// instantiation d'un clip
var monClip:MovieClip = new MovieClip();
```

Nous savons d'après le type `MovieClip` les fonctionnalités disponibles sur une instance de `MovieClip` :

```
// méthodes de la classe MovieClip
monClip.gotoAndPlay(2);
monClip.startDrag();
```

En créant plusieurs instances de `MovieClip`, nous obtenons des objets ayant les mêmes capacités mais des caractéristiques différentes comme par exemple la taille, la position ou encore la transparence :

```
// instantiation d'un premier clip
var monPremierClip:MovieClip = new MovieClip();

// utilisation de l'api de dessin
monPremierClip.graphics.lineStyle ( 1 );
monPremierClip.graphics.beginFill ( 0x880099, 1);
monPremierClip.graphics.drawCircle ( 30, 30, 30 );

// caractéristiques du premier clip
monPremierClip.scaleX = .8;
monPremierClip.scaleY = .8;
monPremierClip.alpha = .5;
monPremierClip.x = 150;
monPremierClip.y = 150;

// affichage du premier clip
addChild ( monPremierClip );

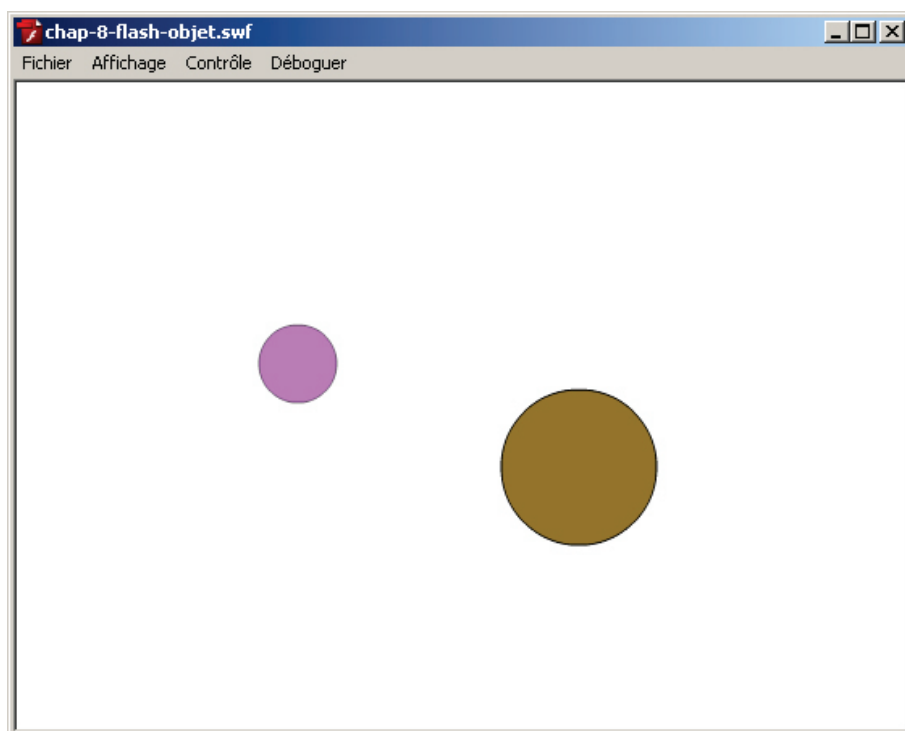
// instantiation d'un second clip
var monSecondClip:MovieClip = new MovieClip();

// utilisation de l'api de dessin
monSecondClip.graphics.lineStyle ( 1 );
monSecondClip.graphics.beginFill ( 0x997711, 1);
monSecondClip.graphics.drawCircle ( 60, 60, 60 );

// caractéristiques du second clip
monSecondClip.scaleX = .8;
monSecondClip.scaleY = .8;
monSecondClip.alpha = 1;
monSecondClip.x = 300;
monSecondClip.y = 190;

// affichage du second clip
addChild ( monSecondClip );
```

Si nous testons le code précédent nous obtenons deux clips de forme circulaire ayant des fonctionnalités communes mais des caractéristiques différentes, comme l'illustre la figure 8-4 :



*Figure 8-4. Deux clips de caractéristiques différentes.*

Au sein de Flash tout est objet, toutes les classes permettent d’instancier des objets ayant des fonctionnalités spécifiques comme la gestion de l’affichage, le chargement de données externe, etc.

Nous allons à présent découvrir comment concevoir nos propres objets en ActionScript 3 afin de les rendre réutilisable et facile d’utilisation. Dans un nouveau fichier ActionScript nous allons créer une classe `Joueur` qui nous permettra de représenter un joueur dans un jeu. En quelques minutes notre classe sera définie et prête à être utilisée.

## A retenir



- Tout est objet.
- A chaque objet une tâche spécifique.
- Les méthodes définissent les capacités d'un objet.
- Les propriétés définissent ses caractéristiques.
- Une instance de classe est un objet créé à partir d'une classe.
- Pour instancier un objet à partir d'une classe, nous utilisons le mot clé `new`.

## Notre première classe

Pour créer un nouveau type d'objet en ActionScript 3 nous devons créer une classe. Il faut considérer celle-ci comme un moule permettant de créer des objets de même type. A partir d'une classe nous pouvons créer autant d'instances de cette classe que nous voulons.

Avant de commencer à coder, il est important de noter que les classes ActionScript 3 résident dans des fichiers externes portant l'extension `.as`. A coté de notre document Flash nous allons définir une classe ActionScript 3 avec un éditeur tel FlashDevelop ou Eclipse. Pour cela nous utilisons le mot clé `class` :

```
class Joueur  
  
{  
  
  
}
```

Nous sauvons cette classe à coté de notre document en cours et nous l'appelons `Joueur.as`. Sur le scénario principal nousinstancions notre joueur avec le mot clé `new` :

```
var monJoueur:Joueur = new Joueur();
```

A la compilation nous obtenons les deux erreurs suivantes :

```
1046: Ce type est introuvable ou n'est pas une constante de compilation :  
Joueur.  
1180: Appel à une méthode qui ne semble pas définie, Joueur.
```

Le compilateur ne semble pas apprécier notre classe, il ne reconnaît pas cette classe que nous venons de définir. Contrairement à ActionScript 2, en ActionScript 3, une classe doit être contenue dans un conteneur de classes appelé paquetage.

## Introduction aux paquetages

Les paquetages (*packages* en anglais) permettent dans un premier temps d'organiser les classes en indiquant dans quel répertoire est située une classe. Nous pouvons imaginer une application dynamique dans laquelle une classe `Connector` serait placée dans un répertoire `serveur`, une autre classe `PNGEncoder` serait elle placée dans un répertoire `encodage`.

Le compilateur trouvera la classe selon le chemin renseigné par le paquetage, afin de définir un paquetage nous utilisons le mot clé `package`.

Si notre classe `Joueur` était placée dans un répertoire `jeu` nous devrions définir notre classe de la manière suivante :

```
package jeu
{
    public class Joueur
    {

    }
}
```

Il serait alors nécessaire de spécifier son chemin au compilateur en utilisant l'instruction `import` avant de pouvoir l'utiliser :

```
import jeu.Joueur;
```

Lorsque notre classe n'est pas placée dans un répertoire spécifique mais réside simplement à côté de notre document FLA aucun import n'est nécessaire, le compilateur vérifie automatiquement à côté du document Flash si les classes existent.

Dans notre cas, la classe `Joueur` est placée à côté du document Flash et ne réside dans aucun répertoire, nous devons donc spécifier un paquetage vide :

```
package
{
    class Joueur
    {

    }
}
```

Dans ce cas, aucun import préalable n'est nécessaire. En ActionScript 2, le mot clé `package` n'existait pas, pour indiquer le chemin d'accès

à une classe nous spécifions le chemin d'accès directement lors de sa déclaration :

```
class jeu.Joueur
{

}

```

La notion de paquetages en ActionScript 3 ne sert pas uniquement à refléter l'emplacement d'une classe, en ActionScript 2, une seule classe pouvait être définie au sein d'un fichier source. En résumé, une seule et unique classe pour chaque fichier .as.

En ActionScript 3, quelques subtilités ont été ajoutées, nous pouvons désormais définir autant de classes que nous souhaitons pour chaque fichier .as. Une seule classe résidera à l'intérieur d'une déclaration de paquetage, les autres devront être définies à l'extérieur de celui-ci et ne pourront pas être utilisées par un code extérieur. Ne vous inquiétez pas, cela paraît relativement abstrait au départ, nous allons au cours des prochains chapitres apprendre à maîtriser la notion de paquetages et découvrir à nouveau de nouvelles fonctionnalités.

Si nous tentons de compiler notre classe à nouveau, les mêmes erreurs s'affichent. Afin de pouvoir instancier notre classe depuis l'extérieur nous devons utiliser l'attribut de classe `public`. Nous allons revenir sur la notion d'attributs de classe très bientôt :

```
package
{
    public class Joueur
    {

    }
}

```

Une fois le mot clé `public` ajouté, la compilation s'effectue sans problème. L'intérêt de notre classe est de pouvoir être instanciée et recevoir différents paramètres lorsque nous utiliserons cette classe dans nos applications. Il y'a de fortes chances qu'un joueur ait un nom, un prénom, ainsi que d'autres propriétés.

Pour instancier notre instance de `Joueur` nous utilisons le mot clé `new` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur();

```

Notre objet `Joueur` est créé pour savoir si un objet correspond à un type spécifique nous utilisons le mot clé `is` :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur ();

// est-ce que notre objet est de type Joueur ?
// affiche : true
trace( monJoueur is Joueur );
```

Le mot clé `instanceof` utilisé en ActionScript 1 et 2 n'existe plus et a été remplacé par le mot clé `is`. En étant de type `Joueur` notre instance nous garantit qu'elle possède toutes les capacités et propriétés propres au type `Joueur`.

### A retenir :

- Pour définir une classe, nous utilisons le mot clé `class`.
- Les classes ActionScript 3 résident dans des fichiers externes portant l'extension `.as`.
- Les classes ActionScript 3 doivent résider dans des paquetages
- Le nom de la classe doit être le même que le fichier `.as`
- Afin d'utiliser une classe nous devons l'importer avec le mot clé `import`.
- Afin de définir un paquetage, nous utilisons le mot clé `package`.
- Pour tester le type d'un objet nous utilisons le mot clé `is`.

### Définition de propriétés

Comme nous l'avons vu précédemment, les propriétés d'un objet décrivent ses caractéristiques. Un être humain peut être considéré comme une instance de la classe `Humain` où chaque caractéristique propre à l'être humain telle la taille, la couleur de ses yeux, ou son nom seraient stockées au sein de propriétés.

Nous allons définir des propriétés au sein de la classe `Joueur` afin que tous les joueurs créés aient leurs propres caractéristiques. Chaque joueur aura un nom, un prénom, une ville d'origine, un âge, et un identifiant les caractérisant. Nous définissons donc cinq propriétés au sein de la classe `Joueur`.

Pour définir des propriétés au sein d'une classe, nous utilisons la définition de classe, un espace situé juste en dessous de la ligne définissant la classe :

```
package
{
    public class Joueur
    {

        // définition des propriétés de la classe
        var nom:String;
```

```
        var prenom:String;  
        var age:int;  
        var ville:String;  
    }  
}
```

En lisant les lignes précédentes nous nous rendons compte que la syntaxe nous rappelle la définition de simples variables.

---

Les propriétés sont en réalité des variables comme les autres, mais évoluant dans le contexte d'un objet.

---

En définissant ces propriétés, nous garantissons que chaque instance de la classe `Joueur` les possède. A tout moment, nous pouvons récupérer ces informations en ciblant les propriétés sur l'instance de classe :

```
// instantiation d'un joueur  
var monJoueur:Joueur = new Joueur();  
  
// récupération du prénom du joueur  
var prenom = monJoueur.prenom;
```

A la compilation le code précédent échoue, car contrairement à ActionScript 2, lorsque nous ne spécifions pas d'attributs pour chaque propriété, celles-ci sont considérées comme non visible depuis l'extérieur du paquetage en cours. Pour gérer l'accès aux propriétés d'une classe nous utilisons les attributs de propriétés.

## Attributs de propriétés de classe

Nous définissons comme membres d'une classe, les propriétés ainsi que les méthodes. Afin de gérer l'accès à chaque membre, nous utilisons des attributs de propriétés.

---

Au sein du modèle d'objet ActionScript les méthodes sont aussi définies par des propriétés. Une méthode n'étant qu'une fonction évoluant dans le contexte d'un objet.

---

Cinq attributs de propriétés de classe existent en ActionScript 3 :

- `internal` : Par défaut, le membre est accessible uniquement depuis le paquetage en cours.
- `public` : Accessible depuis n'importe quelle partie du code.
- `private` : Le membre n'est accessible que depuis la même classe. Les sous-classes n'ont pas accès au membre.
- `protected` : Le membre est accessible depuis la même classe, et les sous-classes.

- **static** : Le membre est accessible uniquement depuis la classe, non depuis les instances.

Nous n'avons pas défini pour le moment d'attributs pour les propriétés de notre classe **Joueur**, lorsqu'aucun attribut n'est défini, la propriété ou méthode est considérée comme **internal** et n'est accessible que depuis une classe appartenant au même paquetage.

Nous reviendrons sur ces subtilités plus tard, pour le moment nous souhaitons accéder à nos propriétés depuis notre scénario nous les définissons comme **public** :

```
package
{
    public class Joueur
    {
        // définition des propriétés publiques de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;
    }
}
```

C'est pour cette raison que nous avons défini notre classe publique afin que celle-ci puisse être instanciée depuis l'extérieur.

Une fois nos propriétés rendues publiques nous pouvons y accéder depuis l'extérieur, nousinstancions un objet **Joueur** puis nous accédons à sa propriété **prenom** :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur();

// récupération du prénom du joueur
var prenom = monJoueur.prenom;

// affiche : null
trace( prenom );
```

En ciblant la propriété **prenom** d'un joueur nous récupérons la valeur **null**. Ce comportement est tout à fait normal, car pour l'instant aucunes données ne sont contenues par les propriétés. Nous les avons simplement définies.

---

En ActionScript 2 le code précédent aurait retourné **undefined** au lieu de **null**.

---

Nous venons de voir qu'à l'aide d'attributs de propriétés de classe nous pouvons gérer l'accès et le comportement des membres d'une

classe. Il existe en ActionScript 3 d'autres attributs liés cette fois aux classes, ces derniers sont appelés *attributs de classe*.

## Attributs de classe

Certaines règles peuvent être appliquées aux classes directement à l'aide de quatre attributs, notons que l'attribut `abstract` n'existe pas en ActionScript 3 :

- `dynamic` : La classe accepte l'ajout de propriétés ou méthodes à l'exécution.
- `final` : La classe ne peut pas être étendue.
- `internal` (par défaut) : La classe n'est accessible que depuis les classes appartenant au même paquetage.
- `public` : La classe est accessible depuis n'importe quel emplacement.

Afin de rendre la classe `Joueur` instanciable depuis l'extérieur nous avons du la rendre publique à l'aide de l'attribut `public`.

Celle-ci est d'ailleurs considérée comme non dynamique, cela signifie qu'il est impossible d'ajouter à l'exécution de nouvelles méthodes ou propriétés à une occurrence de la classe ou à la classe même.

Imaginons que nous souhaitions ajouter une nouvelle propriété appelée `sSexe` à l'exécution :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ();

// ajout d'une nouvelle propriété à l'exécution
premierJoueur.sSexe = "H";
```

Le compilateur se plaint et génère l'erreur suivante :

```
1119: Accès à la propriété sSexe peut-être non définie, via la référence de
type static Joueur.
```

Si nous rendons la classe dynamique à l'aide de l'attribut `dynamic`, l'ajout de membres à l'exécution est possible :

```
package
{
    dynamic public class Joueur
    {
        // définition des propriétés publiques de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;
    }
}
```

```
| }
```

Dans le code suivant nous créons une nouvelle propriété `sSexe` au sein l'instance de classe `Joueur` et récupérons sa valeur :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ();

// ajout d'une nouvelle propriété à l'exécution
premierJoueur.sSexe = "H";

// récupération de la valeur
// affiche : H
trace( premierJoueur.sSexe );
```

Bien que cela soit possible, il est fortement déconseillé d'ajouter de nouveaux membres à l'exécution à une classe ou instance de classe, la lecture de la classe ne reflètera pas les réelles capacités ou caractéristiques de la classe.

En lisant la définition de la classe `Joueur` le développeur pensera trouver cinq propriétés, en réalité une sixième est rajoutée à l'exécution. Ce dernier serait obligé de parcourir tout le code de l'application afin de dénicher toutes les éventuelles modifications apportées à la classe à l'exécution.

Afin d'initialiser notre objet `Joueur` lors de sa création nous devons lui passer des paramètres. Même si notre objet `Joueur` est bien créé, son intérêt pour le moment reste limité car nous ne passons aucun paramètre lors son instanciation. Pour passer des paramètres à un objet lors de son instanciation nous définissons une méthode au sein de la classe appelée *méthode constructeur*.

## Le constructeur

Le but du constructeur est d'initialiser l'objet créé. Celui-ci sera appelé automatiquement dès que la classe sera instanciée.

---

Attention, le constructeur doit impérativement avoir le même nom que la classe. Si ce n'est pas le cas, il sera considéré comme une méthode classique et ne sera pas déclenché.

---

Nous allons prévoir quatre paramètres d'initialisation pour chaque joueur :

- Nom : Une chaîne de caractères représentant son nom.
- Prénom : Une chaîne de caractères représentant son prénom.
- Age : Un entier représentant son âge.
- Ville : Une chaîne de caractères représentant sa location.



Au sein du constructeur nous définissons quatre paramètres, qui serviront à accueillir les futurs paramètres passés :

```
package

{
    public class Joueur
    {

        // définition des propriétés de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
        )

        {

            trace( this );

        }

    }

}
```

Si nous tentons d'instancier notre joueur sans passer les paramètres nécessaires :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur();
```

Le compilateur nous renvoie une erreur :

```
1136: Nombre d'arguments incorrect. 4 attendus.
```

Le compilateur ActionScript 2 ne nous renvoyait aucune erreur lorsque nous tentions d'instancier une classe sans paramètres alors que celle-ci en nécessitait. ActionScript 3 est plus strict et ne permet pas l'instanciation de classes sans paramètres si cela n'est pas spécifié à l'aide du mot clé `rest`. Nous verrons plus tard comment définir une classe pouvant recevoir ou non des paramètres à l'initialisation.

Pour obtenir un joueur, nous créons une instance de la classe `Joueur` en passant les paramètres nécessaires :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// récupération du prénom du joueur
var prenom = monJoueur.prenom;

// affiche : null
trace( prenom );
```

Si nous tentons d'instancier un joueur en passant plus de paramètres que prévus :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan", 50);
```

L'erreur de compilation suivante est générée :

```
1137: Nombre d'arguments incorrect. Nombre maximum attendu : 4.
```

Le compilateur ActionScript 2 était moins strict et permettait de passer un nombre de paramètres en plus de ceux présents au sein de la signature d'une méthode, en ActionScript 3 cela est impossible.

Nous avons instancié notre premier joueur, mais lorsque nous tentons de récupérer son prénom, la propriété `prenom` nous renvoie à nouveau `null`. Nous avons bien passé les paramètres au constructeur, et pourtant nous récupérons toujours `null` lorsque nous ciblons la propriété `prenom`. Est-ce bien normal ?

C'est tout à fait normal, il ne suffit pas de simplement définir un constructeur pour que l'initialisation se fasse comme par magie. C'est à nous de gérer cela, et d'intégrer au sein du constructeur une affectation des paramètres aux propriétés correspondantes. Une fois les paramètres passés, nous devons les stocker au sein de l'objet afin de les mémoriser. C'est le travail du constructeur, ce dernier doit recevoir les paramètres et les affecter à des propriétés correspondantes définies au préalable.

Pour cela nous modifions le constructeur de manière à affecter chaque paramètre passé aux propriétés de l'objet :

```
package
{
    public class Joueur
    {
        // définition des propriétés de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
        )
        {
            // affectation de chaque propriété
            prenom = pPrenom;
            nom = pNom;
            age = pAge;
            ville = pVille;
        }
    }
}
```

```
    }  
  }  
}
```

Une fois les propriétés affectées, nous pouvons récupérer les valeurs associées :

```
// instantiation d'un joueur  
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");  
  
// récupération du prénom du joueur  
var prenom:String = monJoueur.prenom;  
var nom:String = monJoueur.nom;  
var age:int = monJoueur.age;  
var ville:String = monJoueur.ville;  
  
// affiche : Stevie  
trace( prenom );  
// affiche : Wonder  
trace( nom );  
// affiche : 57  
trace( age );  
// affiche : Michigan  
trace( ville );
```

Lorsqu'un objet **Joueur** est instancié nous passons des paramètres d'initialisation, chaque caractéristique est stockée au sein de propriétés.

A l'inverse si nous rendons ces propriétés privées :

```
package  
  
{  
  public class Joueur  
  {  
  
    // définition des propriétés de la classe  
    private var nom:String;  
    private var prenom:String;  
    private var age:int;  
    private var ville:String;  
  
    // fonction constructeur  
    function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String  
  )  
  
    {  
  
      prenom = pPrenom;  
      nom = pNom;  
      age = pAge;  
      ville = pVille;  
  
    }  
  
  }  
}
```

En définissant l'attribut `private`, les propriétés deviennent inaccessibles depuis l'extérieur de la classe :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affiche une erreur à la compilation :
// 1178: Tentative d'accès à la propriété inaccessible nom, via la référence de
type static Joueur.
trace( monJoueur.nom );
```

Lorsque le compilateur est en mode strict, une erreur à la compilation est levée, il est impossible de compiler.

Si nous passons le compilateur en mode non strict, à l'aide de la syntaxe crochet il devient possible de compiler :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");

// provoque une erreur à l'exécution :
trace( monJoueur['nom'] );
```

Mais la machine virtuelle 2 nous rattrape en levant une erreur à l'exécution :

```
ReferenceError: Error #1069: La propriété nom est introuvable sur Joueur et il
n'existe pas de valeur par défaut.
```

Souvenons-nous que la machine virtuelle d'ActionScript 3 (VM2) conserve les types à l'exécution. L'astuce ActionScript 2 consistant à utiliser la syntaxe crochet pour éviter les erreurs de compilation n'est plus valable. Nous verrons tout au long des prochains chapitres, différents cas d'utilisation d'autres attributs de propriétés.

Mais quel serait l'intérêt de définir des propriétés privées ? Quel est l'intérêt si nous ne pouvons pas y accéder ?

Nous allons découvrir très vite d'autres moyens plus sécurisés d'accéder aux propriétés d'un objet. Nous allons traiter cela plus en détail dans quelques instants. Nous venons de découvrir comment définir des propriétés au sein d'une classe ActionScript 3, abordons maintenant la définition de méthodes.

A retenir :

- Le constructeur d'une classe sert à recevoir des paramètres afin d'initialiser l'objet.
- La présence de constructeur n'est pas obligatoire, si omis, le compilateur en définit un par défaut. Dans ce cas, aucun paramètre ne peut être passé lors de l'instanciation de la classe.

### Utilisation du mot clé **this**

Afin de faire référence à *l'objet courant* nous pouvons utiliser le mot clé **this** au sein d'une classe. Nous aurions pu définir la classe **Joueur** de la manière suivante :

```
package  
  
{  
    public class Joueur  
    {  
  
        // définition des propriétés de la classe  
        public var nom:String;  
        public var prenom:String;  
        public var age:int;  
        public var ville:String;  
  
        // fonction constructeur  
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String  
    )  
  
        {  
  
            // affectation de chaque propriété  
            this.prenom = pPrenom;  
            this.nom = pNom;  
            this.age = pAge;  
            this.ville = pVille;  
  
        }  
    }  
}
```

Bien que son utilisation soit intéressante dans certaines situations comme pour passer une référence, son utilisation peut s'avérer redondante et rendre le code verbeux. Certains développeurs apprécient tout de même son utilisation afin de différencier facilement les propriétés d'occurrences des variables locales.

En voyant une variable précédée du mot clé **this**, nous sommes assurés qu'il s'agit d'une propriété membre de la classe. En résumé, il s'agit d'un choix propre à chaque développeur, il n'existe pas de véritable règle d'utilisation.

## Définition de méthodes

Alors que les propriétés d'un objet définissent ses caractéristiques, les méthodes d'une classe définissent ses capacités. Nous parlons généralement de fonctions membres. Dans notre précédent exemple de télévision, les méthodes existantes étaient `allumer`, `eteindre`, `changerChaine`, `baisservolume`, et `monterVolume`.

En appliquant le même concept à notre classe `Joueur` il paraît logique qu'un joueur puisse se présenter, nous allons donc définir une méthode `sePresenter`. Une méthode est un membre régit aussi par les attributs de propriétés. Nous allons rendre cette méthode publique car elle devra être appelée depuis l'extérieur :

```
package
{
    public class Joueur
    {
        // définition des propriétés de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
        )
        {
            prenom = pPrenom;
            nom = pNom;
            age = pAge;
            ville = pVille;
        }

        // méthode permettant au joueur de se présenter
        public function sePresenter ( ):void
        {
            trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
        }
    }
}
```

Pour qu'un joueur se présente nous appelons la méthode `sePresenter` :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");
```

```
// Je m'appelle Stevie, j'ai 57 ans.  
monJoueur.sePresenter();
```

Ainsi, chaque instance de `Joueur` a la capacité de se présenter. La méthode `sePresenter` récupère les propriétés `prenom` et `age` et affiche leurs valeurs. Toutes les propriétés définies au sein de la classe sont accessibles par toutes les méthodes de celle-ci.

Nous allons définir une deuxième méthode nous permettant d'afficher une représentation automatique de l'objet `Joueur`. Par défaut lorsque nous traçons un objet, le lecteur Flash représente l'objet sous une forme simplifiée, indiquant simplement le type de l'objet :

```
// création d'un joueur connu :)  
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );  
  
// affiche : [object Joueur]  
trace( monJoueur );
```

Notons qu'en ActionScript 2, le lecteur affichait simplement `[objet Object]` il fallait ensuite tester à l'aide de l'instruction `instanceof` afin de déterminer le type d'un objet. En ActionScript 3 l'affichage du type est automatique.

En définissant une méthode `toString` au sein de la classe `Joueur` nous redéfinissons la description que le lecteur fait de l'objet :

```
package  
{  
    public class Joueur  
    {  
        // définition des propriétés de la classe  
        public var nom:String;  
        public var prenom:String;  
        public var age:int;  
        public var ville:String;  
  
        // fonction constructeur  
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String  
        )  
        {  
            prenom = pPrenom;  
            nom = pNom;  
            age = pAge;  
            ville = pVille;  
        }  
  
        // méthode permettant au joueur de se présenter  
        public function sePresenter ( ):void  
        {
```

```
        trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
    }

    // affiche une description de l'objet Joueur
    public function toString ( ):String
    {
        return "[Joueur prenom : " + prenom +",  nom : " + nom + " , age : " + age + " , ville : " + ville + " ]";
    }
}
}
```

La méthode `toString` retourne une chaîne de caractères utilisée par le lecteur Flash lorsque celui-ci affiche la description d'un objet. Une fois définie, lorsque nous traçons une instance de la classe `Joueur`, nous obtenons une représentation détaillée de l'objet.

Cela rend notre code plus élégant, lorsqu'un développeur tiers cible une instance de la classe `Joueur`, la description suivante est faite :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affiche : [Joueur prenom : Stevie,  nom : Wonder, age : 57, ville : Michigan]
trace( monJoueur );
```

Dans la quasi-totalité des classes que nous créeront, nous intégrerons une méthode `toString` afin d'assurer une description élégante de nos objets.

Pour le moment nous accédons directement depuis l'extérieur aux propriétés de l'objet `Joueur` car nous les avons définies comme publique. Afin de rendre notre conception plus solide nous allons aborder à présent un nouveau point essentiel de la programmation orientée objet appelé *encapsulation*.

A retenir :



- Les méthodes de la classe ont accès aux propriétés définies au sein de la classe.
- En définissant une méthode `toString` sur nos classes nous modifions la représentation de notre objet faite par le lecteur Flash.

## L'encapsulation

Lorsque nous utilisons une télévision nous utilisons des fonctionnalités sans savoir ce qui se passe en interne, et tant mieux. L'objet nous est livré mettant à disposition des fonctionnalités, nous ne savons rien de ce qui se passe en interne.

Le terme d'encapsulation détermine la manière dont nous exposons les propriétés d'un objet envers le monde extérieur. Un objet bien pensé doit pouvoir être utilisé sans montrer les détails de son *implémentation*. En d'autres termes, un développeur tiers utilisant notre classe `Joueur` n'a pas à savoir que nous utilisons une propriété nommée `age` ou `prenom` ou autres.

Il faut tout d'abord séparer les personnes utilisant les classes en deux catégories :

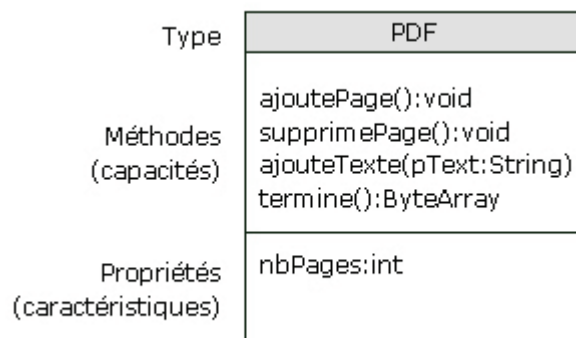
- Le ou les auteurs de la classe.
- Les personnes l'utilisant.

Le code interne à une classe, est appelé *implémentation*. En tant qu'auteur de classes, nous devons impérativement nous assurer de ne pas montrer les détails de l'implémentation de notre classe. Pourquoi ?

Afin d'éviter que la modification d'un détail de l'implémentation n'entraîne de lourdes répercussions pour les développeurs utilisant nos classes.

Prenons un scénario classique, Mathieu, Nicolas et Eric sont développeurs au sein d'une agence Web et développent différentes classes `ActionScript 3` qu'ils réutilisent au cours des différents projets qu'ils doivent réaliser. Nicolas vient de développer une classe permettant de générer des PDF en `ActionScript 3`. Toute l'équipe de développement Flash utilise cette classe dans les nouveaux projets et il faut bien avouer que tout le monde est plutôt satisfait de cette nouvelle librairie.

La figure 8-4 illustre la classe que Nicolas a développée, ainsi que les différentes fonctionnalités disponibles :



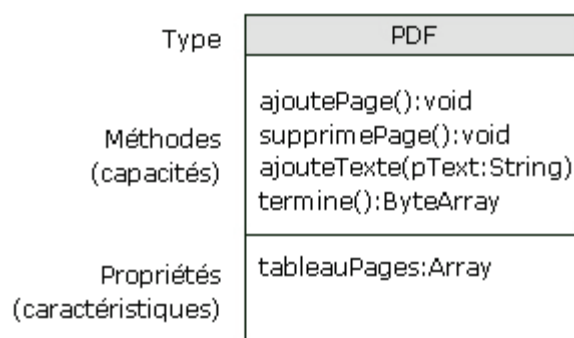
*Figure 8-4. Schéma UML simplifié de la classe PDF.*

La propriété `nbPages` permet de savoir combien de pages comporte le PDF en cours de création. A chaque page créée, Nicolas incrémente la valeur de cette propriété.

Eric vient justement de livrer un projet à un client qui souhaitait générer des PDF depuis Flash. Eric a utilisé la propriété `nbPages` afin de savoir combien de pages sont présentes dans le PDF en cours de création, voici une partie du code d'Eric :

```
// récupération du nombre de pages
var nombresPages:int = monPDF.nbPages;
// affichage du nombre de pages
legende_pages.text = String ( nombresPages );
```

Un matin, Nicolas, auteur de la classe se rend compte que la classe possède quelques faiblesses et décide de rajouter des fonctionnalités. Sa gestion des pages internes au document PDF ne lui paraît pas optimisée, désormais Nicolas stocke les pages dans un tableau accessible par la propriété `tableauPages` contenant chaque page du PDF. Voici la nouvelle version de la classe PDF :



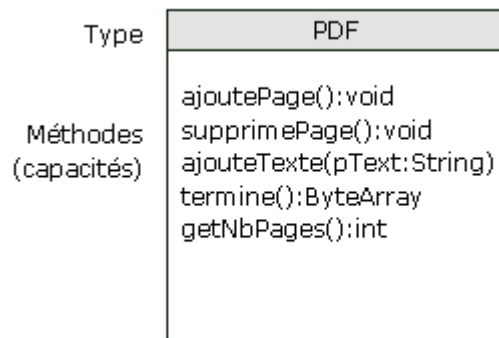
*Figure 8-5. Nouvelle version de la classe PDF.*

Désormais Nicolas utilise un tableau pour stocker chaque page créée. Pour récupérer le nombre de pages, il faut cibler le tableau `tableauPages` et récupérer sa longueur pour connaître le nombre de pages du PDF. Eric et Mathieu récupèrent la nouvelle version mais tous leurs projets ne compilent plus, car la propriété `nbPages` n'existe plus !

Eric et Mathieu auront beau tenter de convaincre Nicolas de remettre la propriété afin que tout fonctionne à nouveau, Nicolas, très content de sa nouvelle gestion interne des pages refuse de rajouter cette propriété `nbPages` qui n'aurait plus de sens désormais.

Qui est le fautif dans ce scénario ?

Nicolas est fautif d'avoir exposé à l'extérieur cette propriété `nbPages`. En modifiant l'implémentation de sa classe, chaque développeur utilisant la classe risque de voir son code obsolète. Nicolas aurait du définir une méthode d'accès `getNbPages` qui peut importe l'implémentation aurait toujours retourné le nombre de pages. Nicolas se rendant compte de sa maladresse, décide de sortir une nouvelle version prenant en compte l'encapsulation. La figure 8-6 illustre la nouvelle classe.



*Figure 8-6. Nouvelle version encapsulée de la classe PDF.*

Eric et Mathieu en appelant la méthode `getNbPages` ne savent pas ce qui se passe en interne et tant mieux ! Nicolas est tranquille et pourra modifier tout ce qu'il souhaite en interne, son seul soucis sera de toujours retourner le nombre de pages quelque soit l'implémentation.

---

Ainsi nous pouvons modifier l'implémentation interne de l'objet sans modifier l'interface de programmation.

---

Nous allons mettre en pratique la notion d'encapsulation et découvrir d'autres avantages liés à ce nouveau concept.

## Mise en pratique de l'encapsulation

Nous venons de voir qu'il est très important de cacher l'implémentation afin de rendre notre conception plus solide. Les parties cachées du code ne sont pas utilisées par les développeurs, ainsi nous sommes libres de le modifier sans entraîner de répercussions négatives. L'encapsulation des propriétés permet ainsi de rendre notre code plus intuitif, nous rendons visible uniquement ce qui est utile et utilisable.

Dans le cas de notre classe `Joueur` nous allons conserver la plupart des propriétés accessibles mais sous une forme différente. Pour accéder à l'âge ou au nom d'un joueur nous accédons directement à des propriétés publiques :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : Stevie
trace( monJoueur.prenom );
// affiche : 57
trace( monJoueur.age );
```

Si un développeur tiers vient à utiliser notre classe `Joueur`, ce dernier pourra utiliser ces deux propriétés. Si nous changeons plus tard l'implémentation de notre classe, ces propriétés pourraient peut être disparaître, ou bien être renommées rendant le code du développeur caduc.

Afin d'éviter tout risque, nous allons définir des méthodes d'accès qui tâcheront de renvoyer la valeur escomptée, quelque soit l'implémentation.

## Les méthodes d'accès

Deux groupes de méthodes d'accès existent :

- Les méthodes récupérant la valeur d'une propriété, appelées méthodes de récupération.
- Les méthodes affectant une valeur à une propriété, appelées méthodes d'affectation.

Prenons un exemple simple, au lieu de cibler directement la propriété `age`, le développeur tiers appellera une méthode `getAge`. Nous rendons dans un premier temps toutes nos propriétés privées afin que celles-ci soit cachées :

```
| package
```

```
{  
  
    public class Joueur  
    {  
  
        // définition des propriétés de la classe  
        private var nom:String;  
        private var prenom:String;  
        private var age:int;  
        private var ville:String;  
  
        // fonction constructeur  
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String  
    )  
  
        {  
  
            prenom = pPrenom;  
            nom = pNom;  
            age = pAge;  
            ville = pVille;  
  
        }  
  
        // méthode permettant au joueur de se présenter  
        public function sePresenter ( ):void  
  
        {  
  
            trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );  
  
        }  
  
        // affiche une description de l'objet Joueur  
        public function toString ( ):String  
  
        {  
  
            return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :  
" + age + ", ville : " + ville + "];"  
  
        }  
  
    }  
  
}
```

En programmation orientée objet, il est fortement conseillé de protéger les propriétés en les rendant privées. Puis nous définissons une méthode `getAge` afin de récupérer l'âge du joueur :

```
package  
  
{  
  
    public class Joueur  
    {  
  
        // définition des propriétés de la classe  
        private var nom:String;  
        private var prenom:String;
```

```
private var age:int;
private var ville:String;

// fonction constructeur
function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

{

    prenom = pPrenom;
    nom = pNom;
    age = pAge;
    ville = pVille;

}

// récupère l'age du joueur
public function getAge ( ):int

{

    return age;

}

// méthode permettant au joueur de se présenter
public function sePresenter ( ):void

{

    trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

}

// affiche une description de l'objet Joueur
public function toString ( ):String

{

    return "[Joueur prenom : " + prenom + ", nom : " + nom + ", age : " + age + ", ville : " + ville + "];"

}

}
```

Si nous tentons d'accéder aux propriétés privées, le compilateur empêche de compiler :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : 1178: Tentative d'accès à la propriété inaccessible prenom, via la
référence de type static Joueur.
trace( monJoueur.prenom );

// affiche : 1178: Tentative d'accès à la propriété inaccessible age, via la
référence de type static Joueur.
trace( monJoueur.age );
```

Désormais, afin de récupérer l'âge d'un joueur nous appelons la méthode `getAge` :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : 57
trace( monJoueur.getAge() );
```

En utilisant des méthodes d'accès nous cachons l'implémentation de la classe et rendons les modifications futures sécurisées et plus faciles. Les développeurs ne savent pas que la propriété `age` est utilisée en interne. Si nous décidons de changer l'implémentation pour des raisons d'optimisations ou de conception, aucun risque pour les développeurs utilisant la classe.

Dans l'exemple suivant les propriétés du joueur sont désormais stockées au sein d'un objet :

```
package
{
    public class Joueur
    {
        // définition des propriétés de la classe
        private var infosJoueur:Object;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
        )
        {
            // les paramètres sont désormais stockés
            // dans un objet infosJoueur
            infosJoueur = new Object();
            // chaque paramètre est stocké dans l'objet infoJoueurs
            infosJoueur.prenom = pPrenom;
            infosJoueur.nom = pNom;
            infosJoueur.age = pAge;
            infosJoueur.ville = pVille;
        }

        // récupère l'age du joueur
        public function getAge ( ):int
        {
            //récupère la propriété age au sein de l'objet infosJoueur
            return infosJoueur.age;
        }

        // méthode permettant au joueur de se présenter
        public function sePresenter ( ):void
```

```
        {
            trace("Je m'appelle " + infosJoueur.prenom + ", j'ai " +
infosJoueur.age + " ans." );
        }

        // affiche une description de l'objet Joueur
        public function toString ( ):String
        {
            return "[Joueur prenom : " + infosJoueur.prenom +",  nom : " +
infosJoueur.nom + ", age : " + infosJoueur.age + ", ville : " +
infosJoueur.ville + "]";
        }
    }
}
```

L'accès aux propriétés ne change pas, notre code continue de fonctionner :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : 57
trace( monJoueur.getAge() );
```

Ainsi, notre implémentation change sans intervenir sur l'interface de programmation.

Il est important de signaler qu'une méthode de récupération peut porter n'importe quel nom, l'utilisation du mot `get` au sein de la méthode `getAge` n'est pas obligatoire. Nous aurions pu appeler cette même méthode `recupAge`.

Nous définissons une méthode de récupération spécifique à chaque propriété :

```
package
{
    public class Joueur
    {
        // définition des propriétés de la classe
        private var nom:String;
        private var prenom:String;
        private var age:int;
        private var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
        )
```



```
{
    prenom = pPrenom;
    nom = pNom;
    age = pAge;
    ville = pVille;
}

// récupère le prénom du joueur
public function getPrenom ( ):String
{
    return prenom;
}

// récupère le nom du joueur
public function getNom ( ):String
{
    return nom;
}

// récupère l'age du joueur
public function getAge ( ):int
{
    return age;
}

// récupère la ville du joueur
public function getVille ( ):String
{
    return ville;
}

// méthode permettant au joueur de se presenter
public function sePresenter ( ):void
{
    trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
}

// affiche une description de l'objet Joueur
public function toString ( ):String
{
    return "[Joueur prenom : " + prenom + ", nom : " + nom + ", age : " + age + ", ville : " + ville + "];"
```

```
    }  
  }  
}
```

De cette manière, nous pouvons accéder à toutes les propriétés d'une instance de **Joueur** :

```
// création d'un joueur connu :)  
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );  
  
// affiche : Stevie  
trace( monJoueur.getPrenom() );  
  
// affiche : Wonder  
trace( monJoueur.getNom() );  
  
// affiche : 57  
trace( monJoueur.getAge() );  
  
// affiche : Michigan  
trace( monJoueur.getVille() );
```

Grace aux différentes méthodes de récupération les propriétés privées sont rendues accessibles de manière encapsulée. Pour l'instant il est impossible de les modifier, car nous n'avons pas encore intégré de méthodes d'affectation.

## Contrôle d'affectation

Nous venons de voir comment rendre notre code encapsulé grâce aux méthodes de récupération et d'affectation, l'autre grand intérêt de l'encapsulation concerne le contrôle d'affectation des propriétés.

Sans méthodes d'affectation, il est impossible de rendre l'affectation ou l'accès à nos propriétés intelligentes. Imaginons que le nom de la ville doive toujours être formaté correctement. Pour cela nous ajoutons une méthode d'accès **setVille** qui intègre une logique spécifique :

```
package  
{  
    public class Joueur  
    {  
        // définition des propriétés de la classe  
        private var nom:String;  
        private var prenom:String;  
        private var age:Number;  
        private var ville:String;  
  
        // fonction constructeur  
        function Joueur ( pPrenom:String, pNom:String, pAge:Number,  
pVille:String )
```

```
{
    prenom = pPrenom;
    nom = pNom;
    age = pAge;
    ville = pVille;
}

// récupère le prénom du joueur
public function getPrenom ( ):String
{
    return prenom;
}

// récupère le nom du joueur
public function getNom ( ):String
{
    return nom;
}

// récupère l'age du joueur
public function getAge ( ):Number
{
    return age;
}

// récupère la ville du joueur
public function getVille ( ):String
{
    return ville;
}

// récupère la ville du joueur
public function setVille ( pVille:String ):void
{
    ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
).toLowerCase();
}

// méthode permettant au joueur de se presenter
public function sePresenter ( ):void
{
    trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
}
```

```
    }

    // affiche une description de l'objet Joueur
    public function toString ( ):String

    {

        return "[Joueur prenom : " + prenom + ", nom : " + nom + ", age : " + age + ", ville : " + ville + "]";

    }

}
```

Le changement de ville est désormais contrôlé, si un nom de ville est passé, le formatage est automatique :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affectation d'une ville ma formatée
monJoueur.setVille ( "CLEEVELAnd");

// affiche : Cleveland
trace( monJoueur.getVille() );
```

La méthode `setVille` formate automatiquement la ville passée au format attendu. En définissant cette méthode, nous sommes assurés du formatage des villes associées à chaque joueur. Pratique n'est-ce pas ?

Afin d'éviter les erreurs d'exécution, nous pouvons affecter les propriétés d'un objet en exerçant un contrôle. Nous rendons notre objet intelligent.

Une autre situation pourrait être imaginée. Imaginons que l'application serveur gérant la connexion de nos joueurs ne puisse gérer des noms ou prénoms de plus de 30 caractères, nous intégrons ce contrôle au sein des méthodes `setNom` et `setPrenom`. Nous rajoutons au passage une méthode `setAge` arrondissant automatiquement l'âge passé :

```
package
{

    public class Joueur
    {

        // définition des propriétés de la classe
        private var nom:String;
        private var prenom:String;
        private var age:Number;
        private var ville:String;

        // fonction constructeur
```

```
function Joueur ( pPrenom:String, pNom:String, pAge:Number,
pVille:String )

{

    prenom = pPrenom;
    nom = pNom;
    age = pAge;
    ville = pVille;

}

// récupère le prénom du joueur
public function getPrenom ( ):String

{

    return prenom;

}

// récupère le nom du joueur
public function getNom ( ):String

{

    return nom;

}

// récupère l'age du joueur
public function getAge ( ):Number

{

    return age;

}

// récupère la ville du joueur
public function getVille ( ):String

{

    return ville;

}

// permet de changer le prénom du joueur
public function setPrenom ( pPrenom:String ):void

{

    if ( pPrenom.length <= 30 ) prenom = pPrenom;

    else trace ("Le prénom spécifié est trop long");

}

// permet de changer le nom du joueur
public function setNom ( pNom:String ):void
```

```
    {  
        if ( pNom.length <= 30 ) nom = pNom;  
        else trace ("Le nom spécifié est trop long");  
    }  
  
    // permet de changer l'âge du joueur  
    public function setAge ( pAge:Number ):void  
    {  
        // l'age passé est automatiquement arrondi  
        age = Math.floor ( pAge );  
    }  
  
    // récupère la ville du joueur  
    public function setVille ( pVille:String ):void  
    {  
        ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1  
).toLowerCase();  
    }  
  
    // méthode permettant au joueur de se présenter  
    public function sePresenter ( ):void  
    {  
        trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );  
    }  
  
    // affiche une description de l'objet Joueur  
    public function toString ( ):String  
    {  
        return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :  
" + age + ", ville : " + ville + "];"  
    }  
}  
}
```

Si un nom ou un prénom trop long est passé nous pouvons afficher un message avertissant le développeur tiers :

```
// création d'un joueur connu :)  
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");  
  
// affectation du nouveau nom  
// affiche : Le nom spécifié est trop long  
monJoueur.setNom ( "UnNomTresTresTresTresTresTresLong" );  
  
// affiche : Wonder
```

```
trace( monJoueur.getNom() );

// affectation d'un nouvel age
monJoueur.setAge ( 24.8 );

// affiche : 24
trace( monJoueur.getAge() );
```

En effectuant ce contrôle, nous sommes assurés que la propriété `nom` ne contiendra aucune chaîne de caractères de plus de 30 caractères. Sans cette vérification faite au moment de l’affectation, nous serions obligés de tester la propriété `nom` avant de faire quoi que ce soit.

Les méthodes d’affectation et de récupération s’avèrent très pratiques, elles offrent la possibilité de délivrer un code sécurisé, portable et élégant. Pourtant certains développeurs préfèrent utiliser les méthodes en lecture/écriture qu’ils considèrent comme plus pratiques.

L’utilisation d’une méthode pour récupérer ou affecter une propriété ne leur convient pas. `ActionScript 3` propose une solution alternative, appelées méthodes en lecture/écriture.

## Méthodes en lecture/écriture

Les méthodes de lecture et d’écriture plus communément appelées *getter/setter* sont la prolongation des méthodes d’accès. Leur intérêt reste le même, elles permettent d’encapsuler notre code en gérant l’affectation et la récupération de propriétés d’un objet mais à l’aide d’une syntaxe différente.

Les méthodes de lecture et d’écriture permettent au développeur d’appeler de manière transparente ces méthodes comme si ce dernier ciblait une propriété. Afin de bien comprendre ce concept nous allons intégrer des méthodes de lecture et d’écriture dans notre classe `Joueur`.

Afin de définir ces méthodes nous utilisons deux mots clés :

- `get` : Définit une méthode de lecture
- `set` : Définit une méthode d’écriture

Une méthode de lecture se définit de la manière suivante :

```
// affecte une propriété
public function get maPropriete ( ):type

{

    return proprieteInterne;

}
```

Celle-ci doit obligatoirement retourner une valeur, et ne posséder aucun paramètre au sein de sa signature. Si ce n'est pas le cas, une erreur à la compilation est générée.

Une méthode d'écriture se définit de la manière suivante :

```
// affecte une propriété
public function set maPropriete ( pNouvelleValeur:type ):void
{
    proprieteInterne = pNouvelleValeur;
}
```

Celle-ci doit obligatoirement posséder au moins un paramètre au sein de sa signature et ne retourner aucune valeur. Si ce n'est pas le cas, une erreur à la compilation est générée.

Nous allons modifier les méthodes de récupération et d'affectation `setNom` et `getNom` par des méthodes en lecture/écriture.

Afin d'éviter un conflit de définition de variables, nous devons nous assurer que les méthodes en lecture/écriture ne possèdent pas le même nom que les propriétés que nous souhaitons externaliser :

```
package
{
    public class Joueur
    {
        // définition des propriétés de la classe
        private var _nom:String;
        private var prenom:String;
        private var age:Number;
        private var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:Number,
pVille:String )
        {
            prenom = pPrenom;
            _nom = pNom;
            age = pAge;
            ville = pVille;
        }

        // récupère le prénom du joueur
        public function getPrenom ( ):String
        {
            return prenom;
        }
    }
}
```



```
}

// récupère l'age du joueur
public function getAge ( ):Number

{

    return age;

}

// récupère la ville du joueur
public function getVille ( ):String

{

    return ville;

}

// permet de changer le prénom du joueur
public function setPrenom ( pPrenom:String ):void

{

    if ( pPrenom.length <= 30 ) prenom = pPrenom;

    else trace ("Le prénom spécifié est trop long");

}

// récupère le nom du joueur
public function get nom ( ):String

{

    return _nom;

}

// permet de changer le nom du joueur
public function set nom ( pNom:String ):void

{

    if ( pNom.length <= 30 ) _nom = pNom;

    else trace ("Le nom spécifié est trop long");

}

// permet de changer l'age du joueur
public function setAge ( pAge:Number ):void

{

    // l'age passé est automatiquement arrondi
    age = Math.floor ( pAge );

}
```

```

        // récupère la ville du joueur
        public function setVille ( pVille:String ):void
        {
            ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
        ).toLowerCase();
        }

        // méthode permettant au joueur de se présenter
        public function sePresenter ( ):void
        {
            trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
        }

        // affiche une description de l'objet Joueur
        public function toString ( ):String
        {
            return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
        " + age + ", ville : " + ville + "]";
        }
    }
}

```

En utilisant le mot clé **set** nous avons défini une méthode d’écriture. Ainsi, le développeur à l’impression d’affecter une propriété, en réalité notre méthode d’écriture est déclenchée :

```

// instantiation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affectation du nouveau nom
monJoueur.nom = "Womack";

```

Cela permet de conserver une affectation des données, dans la même écriture que les propriétés, en conservant le contrôle des données passées :

```

// instantiation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affectation du nouveau nom
// affiche : Le nom spécifié est trop long
monJoueur.nom = "UnNomTresTresTresTresTresTresLong";

```

En utilisant le mot clé **get** nous avons défini une méthode de lecture.

Ainsi, le développeur à l’impression de cibler une simple propriété, en réalité notre méthode de lecture est déclenchée :

```

// instantiation d'un joueur

```

```
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affectation du nouveau nom
// affiche : Le nom spécifié est trop long
monJoueur.nom = "UnNomTresTresTresTresTresTresLong";

// récupération du nom
// affiche : Wonder
trace( monJoueur.nom ) ;
```

Voici le code final de notre classe **Joueur** :

```
package

{

    public class Joueur
    {

        // définition des propriétés de la classe
        private var _nom:String;
        private var _prenom:String;
        private var _age:Number;
        private var _ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:Number,
pVille:String )

        {

            _prenom = pPrenom;
            _nom = pNom;
            _age = pAge;
            _ville = pVille;

        }

        // récupère le nom du joueur
        public function get nom ( ):String

        {

            return _nom;

        }

        // permet de changer le nom du joueur
        public function set nom ( pNom:String ):void

        {

            if ( pNom.length <= 30 ) _nom = pNom;

            else trace ("Le nom spécifié est trop long");

        }

        // récupère le prénom du joueur
        public function get prenom ( ):String

        {
```

```
        return _prenom;
    }

    // permet de changer le prénom du joueur
    public function set prenom ( pPrenom:String ):void
    {
        if ( pPrenom.length <= 30 ) _prenom = pPrenom;
        else trace ("Le prénom spécifié est trop long");
    }

    // permet de changer l'age du joueur
    public function set age ( pAge:Number ):void
    {
        // l'age passé est automatiquement arrondi
        _age = Math.floor ( pAge );
    }

    // récupère l'age du joueur
    public function get age ( ):Number
    {
        return _age;
    }

    // récupère la ville du joueur
    public function set ville ( pVille:String ):void
    {
        _ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
    ).toLowerCase();
    }

    // récupère la ville du joueur
    public function get ville ( ):String
    {
        return _ville;
    }

    // méthode permettant au joueur de se présenter
    public function sePresenter ( ):void
    {
        trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
    }
}
```

```
// affiche une description de l'objet Joueur
public function toString ( ):String

{

    return "[Joueur prenom : " + prenom + ",  nom : " + nom + ", age : "
    + age + ", ville : " + ville + "];"

}

}
```

L'encapsulation fait partie des points essentiels de la programmation orientée objet, les méthodes de récupération et d'affectation permettent de cacher l'implémentation et d'exercer un contrôle lors de l'affectation de propriétés. Les méthodes de lecture/écriture étendent ce concept en proposant une syntaxe alternative.

Libre à vous de choisir ce que vous préférez, de nombreux débats existent quand au choix d'utilisateur de méthodes de récupération et d'affectation et de lecture/écriture.

## A retenir

- Dans la plupart des cas, pensez à rendre privées les propriétés d'un objet.
- Afin d'encapsuler notre code nous pouvons utiliser des méthodes de récupération ou d'affectation ou bien des méthodes de lecture et d'écriture.
- L'intérêt est d'exercer un contrôle sur la lecture et l'écriture de propriétés.
- Un code encapsulé rend la classe évolutive et solide.

## Cas d'utilisation de l'attribut static

L'attribut de propriété `static` permet de rendre un membre utilisable dans un contexte de classe et non d'occurrence. En d'autres termes, lorsqu'une méthode ou une propriété est définie avec l'attribut `static` celle-ci ne peut être appelée que depuis le constructeur de la classe.

Les propriétés et méthodes statiques s'opposent aux méthodes et propriétés d'occurrences qui ne peuvent être appelées que sur des instances de classes. L'intérêt d'une méthode ou propriété statique est d'être globale à la classe, si nous définissons une propriété statique `nVitesse` au sein d'une classe `Vaisseau`, lors de sa modification tout les vaisseaux voient leur vitesse modifiée.

Parmi les méthodes statiques les plus connues nous pouvons citer :

- `Math.abs`
- `Math.round`
- `Math.floor`
- `ExternalInterface.call`

Nous utilisons très souvent ses propriétés statiques au sein de classes comme `Math` ou `System` :

- `Math.PI`
- `System.totalMemory`
- `Security.sandboxType`

Voici un exemple canonique d'utilisation d'une propriété statique. Afin de savoir combien d'instances de classes ont été créées, nous définissons une propriété statique `i` au sein de la classe `Joueur`.

A chaque instantiation nous incrémentons cette propriété et affectons la valeur à la propriété d'occurrence `id` :

```
package
{
    public class Joueur
    {
        // définition des propriétés de la classe
        private var nom:String;
        private var prenom:String;
        private var age:Number;
        private var ville:String;

        // propriété statique
        private static var i:int = 0;

        // propriété id
        private var id:int;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
        )
        {
            prenom = pPrenom;
            nom = pNom;
            age = pAge;
            ville = pVille;

            // à chaque objet Joueur créé, nous incrémentons
            // la propriété statique i
            id = i++;
        }
    }
}
```

```
    }  
  
    // reste du code de la classe non montré  
}  
}
```

A chaque instanciation d'un objet `Joueur`, le constructeur est déclenché et incrémente la propriété statique `i` aussitôt affectée à la propriété d'occurrence `id`.

Pour pouvoir récupérer le nombre de joueurs créés, nous allons créer une méthode statique nous renvoyant la valeur de la propriété statique `i` :

```
// renvoie le nombre de joueurs créés  
public static function getNombreJoueurs ( ):int  
  
{  
  
    return i;  
}
```

Afin de cibler une propriété statique au sein d'une classe nous précisons par convention toujours le constructeur de la classe avant de cibler la propriété :

```
// fonction constructeur  
function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String )  
  
{  
  
    prenom = pPrenom;  
    nom = pNom;  
    age = pAge;  
    ville = pVille;  
  
    id = Joueur.i++;  
}  
  
// renvoie le nombre de joueurs créés  
public static function getNombreJoueurs ( ):int  
  
{  
  
    return Joueur.i;  
}
```

En spécifiant le constructeur avant la propriété, un développeur tiers serait tout de suite averti de la nature statique d'une propriété.

Voici le code final de la classe `Joueur` :

```
package
{
    public class Joueur
    {
        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
        )

        {

            prenom = pPrenom;
            nom = pNom;
            age = pAge;
            ville = pVille;

            id = Joueur.i++;

        }

        // renvoie le nombre de joueurs créés
        public static function getNombreJoueurs ( ):int

        {

            return Joueur.i;

        }

        // récupère le prénom du joueur
        public function getPrenom ( ):String

        {

            return prenom;

        }

        // récupère le nom du joueur
        public function getNom ( ):String

        {

            return nom;

        }

        // récupère l'age du joueur
        public function getAge ( ):int

        {

            return age;

        }

        // récupère la ville du joueur
        public function getVille ( ):String

        {
```



```
        return ville;
    }

    // récupère la ville du joueur
    public function setVille ( pVille:String ):void
    {
        ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
    ).toLowerCase();
    }

    // permet de changer le nom du joueur
    public function setNom ( pNom:String ):void
    {
        if ( pNom.length <= 30 ) nom = pNom;
        else trace ("Le nom spécifié est trop long");
    }

    // permet de changer le prénom du joueur
    public function setPrenom ( pPrenom:String ):void
    {
        if ( pPrenom.length <= 30 ) prenom = pPrenom;
        else trace ("Le prénom spécifié est trop long");
    }

    // méthode permettant au joueur de se présenter
    public function sePresenter ( ):void
    {
        trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
    }

    // affiche une description de l'objet Joueur
    public function toString ( ):String
    {
        return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
    " + age + ", ville : " + ville + "]\n";
    }
}
}
```

Afin de savoir combien de joueurs ont été créés, nous appelons la méthode `getNombreJoueurs` directement depuis la classe `Joueur` :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : 1
trace( Joueur.getNombreJoueurs() );

// instantiation d'un joueur
var monSecondJoueur:Joueur = new Joueur("Bobby", "Womack", 57, "Detroit");

// affiche : 2
trace( Joueur.getNombreJoueurs() );
```

Si nous tentons d'appeler une méthode de classe sur une occurrence de classe :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");

// appel d'une méthode statique sur une instance de classe
monJoueur.getNombreJoueurs();
```

Le compilateur nous renvoie l'erreur suivante :

```
1061: Appel à la méthode getNombreJoueurs peut-être non définie, via la
référence de type static Joueur.
```

Contrairement aux propriétés d'occurrences de classes, les propriétés de classes peuvent être initialisées directement après leur définition. Ceci est dû au fait qu'une méthode statique n'existe qu'au sein d'une classe et non des occurrences, ainsi si nous créons dix joueurs le constructeur de la classe `Joueur` sera déclenchée dix fois, mais l'initialisation de la classe et de ses propriétés statiques une seule fois. Ainsi notre tableau `tableauJoueurs` n'est pas recréé à chaque déclenchement du constructeur.

Il est impossible d'utiliser le mot clé `this` au sein d'une méthode de classe, seules les méthodes d'instances le permettent. Une méthode statique évolue dans le contexte d'une classe et non d'une instance. Si nous tentons de référencer `this` au sein d'une méthode statique, la compilation est impossible :

```
// renvoie le nombre de joueurs créés
public static function getNombreJoueurs ( ):int
{
    return this.i;
}
```

Le message d'erreur suivant s'affiche :

```
1042: Il est impossible d'utiliser le mot-clé this dans une méthode statique.
Il ne peut être utilisé que dans les méthodes d'une instance, la fermeture
d'une fonction et le code global.
```

A l'inverse les méthodes d'instances peuvent cibler une propriété ou une méthode statique. Le constructeur de la classe `Joueur` incrémente la propriété statique `i` :

```
// fonction constructeur
function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

{

    prenom = pPrenom;
    nom = pNom;
    age = pAge;
    ville = pVille;

    // A chaque objet Joueur créé, nous incrémentons la propriété
    statique i
    id = Joueur.i++;

}
```

Lorsque nous avons besoin de définir une propriété ou une méthode ayant une signification globale à toutes les instances de classes nous utilisons l'attribut de propriétés `static`. Certaines classes ne contiennent que des méthodes statiques, nous pourrions imaginer une classe `VerifFormulaire` disposant de différentes méthodes permettant de valider un formulaire.

Au lieu de créer une instance de classe puis d'appeler la méthode sur celle-ci, nous appelons directement la méthode `verifEmail` sur la classe :

```
// vérification du mail directement à l'aide d'une méthode statique
var mailValide:Boolean = OutilsFormulaire.verifEmail( "bobby@groove.com" );
```

Cette technique permet par exemple la création de classes utilitaires, rendant l'accès à des fonctionnalités sans instancier d'objet spécifique. Nous piochons directement sur la classe la fonctionnalité qui nous intéresse.

### A retenir :

- L'utilisation du mot clé `this` est interdite au sein d'une méthode statique.
- A l'inverse, une méthode d'instance de classe peut cibler ou appeler une méthode ou propriété de classe.
- Les méthodes ou propriétés statiques ont un sens global à la classe.

## La classe `JoueurManager`

La programmation objet prend tout son sens lorsque nous faisons travailler plusieurs objets ensemble, nous allons maintenant créer une

classe qui aura comme simple tâche d'ajouter ou supprimer nos joueurs au sein de l'application, d'autres fonctionnalités comme le tri ou autres pourront être ajoutées plus tard.

Le but de la classe `JoueurManager` sera de centraliser la gestion des joueurs de notre application. A côté de notre classe `Joueur` nous définissons une nouvelle classe appelée `JoueurManager` :

```
package
{
    public class JoueurManager
    {
        public function JoueurManager ( )
        {

        }

    }
}
```

Nous définissons une propriété d'occurrence `tableauJoueurs` afin de contenir chaque joueur :

```
package
{
    public class JoueurManager
    {
        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array = new Array();

        public function JoueurManager ( )
        {

        }

    }
}
```

La méthode `ajouteJoueur` ajoute la référence au joueur passé en paramètre au tableau interne `tableauJoueurs` et appelle la méthode `sePresenter` sur chaque joueur entrant :

```
package
```

---

```
{  
  
    public class JoueurManager  
    {  
  
        // tableau contenant les références de joueurs  
        private var tableauJoueurs:Array = new Array();  
  
        public function JoueurManager ( )  
        {  
  
        }  
  
        public function ajouteJoueur ( pJoueur:Joueur ):void  
        {  
  
            pJoueur.sePresenter();  
  
            tableauJoueurs.push ( pJoueur );  
  
        }  
    }  
}
```

La méthode `ajouteJoueur` accepte un paramètre de type `Joueur`, seuls des objets de type `Joueur` pourront donc être passés. Lorsqu'un joueur est ajouté, la méthode `sePresenter` est appelée sur celui-ci.

Ainsi, pour ajouter des joueurs nous créons chaque instance puis les ajoutons au gestionnaire de joueurs, la classe `JoueurManager` :

```
// instantiation d'un joueur  
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");  
var monDeuxiemeJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");  
var monTroisiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48, "Los Angeles");  
  
// gestion des joueurs  
var monManager:JoueurManager = new JoueurManager();  
  
// ajout des joueurs  
// affiche :  
/*  
Je m'appelle Stevie, j'ai 57 ans.  
Je m'appelle Bobby, j'ai 66 ans.  
Je m'appelle Michael, j'ai 48 ans.  
*/  
monManager.ajouteJoueur ( monJoueur );  
monManager.ajouteJoueur ( monDeuxiemeJoueur );  
monManager.ajouteJoueur ( monTroisiemeJoueur );
```

Lorsqu'un joueur est ajouté, il se présente automatiquement. Au cas où un joueur serait supprimé par le système, nous devons le supprimer du gestionnaire, nous définissons une méthode `supprimeJoueur`.

Celle-ci recherche le joueur dans le tableau interne grâce à la méthode `indexOf` et le supprime si celui-ci est trouvé. Autrement, nous affichons un message indiquant que le joueur n'est plus présent dans le gestionnaire :

```
package
{
    public class JoueurManager
    {
        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array = new Array();

        public function JoueurManager ( )
        {

        }

        public function ajouteJoueur ( pJoueur:Joueur ):void
        {
            pJoueur.sePresenter();
            tableauJoueurs.push ( pJoueur );
        }

        public function supprimeJoueur ( pJoueur:Joueur ):void
        {
            var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

            if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

            else trace("Joueur non présent !");
        }
    }
}
```

Nous pouvons supprimer un joueur en passant sa référence. Afin d'externaliser le tableau contenant tout les joueurs, nous définissons une méthode d'accès `getJoueurs` :

```
package
{

    public class JoueurManager
    {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array;

        public function JoueurManager ( )

        {

            tableauJoueurs = new Array();

        }

        public function ajouteJoueur ( pJoueur:Joueur ):void
        {

            pJoueur.sePresenter();

            tableauJoueurs.push ( pJoueur );

        }

        public function supprimeJoueur ( pJoueur:Joueur ):void
        {

            var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

            if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

            else trace("Joueur non présent !");

        }

        public function getJoueurs ( ):Array
        {

            return tableauJoueurs;

        }

    }

}
```

Afin de récupérer l'ensemble des joueurs, nous appelons la méthode `getJoueurs` sur l'instance de classe `JoueurManager` :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");
var monDeuxiemeJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monTroisiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48, "Los
Angeles");

// gestion des joueurs
```

```
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs
monManager.ajouteJoueur ( monJoueur );
monManager.ajouteJoueur ( monDeuxiemeJoueur );
monManager.ajouteJoueur ( monTroisiemeJoueur );

// récupération de l'ensemble des joueurs
// affiche : [Joueur prenom : Stevie, nom : Wonder, age : 57, ville :
Michigan],[Joueur prenom : Bobby, nom : Womack, age : 66, ville :
Detroit],[Joueur prenom : Michael, nom : Jackson, age : 48, ville : Los
Angeles]
trace( monManager.getJoueurs() );
```

Afin de supprimer des joueurs nous appelons la méthode `supprimeJoueur` en passant le joueur à supprimer en référence :

```
// instantiation d'un joueur
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");
var monDeuxiemeJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monTroisiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48, "Los
Angeles");

// gestion des joueurs
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs
monManager.ajouteJoueur ( monJoueur );
monManager.ajouteJoueur ( monDeuxiemeJoueur );
monManager.ajouteJoueur ( monTroisiemeJoueur );

// suppression de deux joueurs
monManager.supprimeJoueur ( monJoueur );
monManager.supprimeJoueur ( monTroisiemeJoueur );

// récupération de l'ensemble des joueurs
// affiche : [Joueur prenom : Bobby, nom : Womack, age : 66, ville :
Detroit]
trace( monManager.getJoueurs() );
```

La classe `JoueurManager` nous permet de gérer les différents joueurs créés, nous pourrions ajouter de nombreuses fonctionnalités. En programmation orientée objet il n'existe pas une seule et unique manière de concevoir chaque application. C'est justement la richesse du développement orienté objet, nous pouvons discuter des heures durant, de la manière dont nous avons développé une application, certains seront d'accord avec votre raisonnement d'autres ne le seront pas. L'intérêt est d'échanger idées et point de vue sur une conception.

Lorsque nous devons développer une application, nous sommes souvent confrontés à des problèmes qu'un autre développeur a sûrement rencontrés avant nous. Pour apporter des solutions concrètes à des problèmes récurrents nous pouvons utiliser des *modèles de conceptions*. Ces derniers définissent la manière dont nous devons séparer et concevoir notre application toujours dans un objectif d'optimisation, de portabilité et de réutilisation. Nous reviendrons bientôt sur les modèles de conception les plus utilisés.



Nos classes `Joueur` et `JoueurManager` sont désormais utilisées dans nos applications, mais le client souhaite introduire la notion de modérateurs. En y réfléchissant quelques instants nous pouvons facilement admettre qu'un modérateur est une sorte de joueur, mais disposant de droits spécifiques comme le fait d'exclure un autre joueur ou de stopper une partie en cours. Un administrateur possède donc toutes les capacités d'un joueur. Il serait redondant dans ce cas précis, de redéfinir toutes les méthodes et propriétés déjà définies au sein de la classe `Joueur` au sein de la classe `Administrateur`, pour réutiliser notre code nous allons utiliser l'héritage.

## L'héritage

La notion d'héritage est un concept clé de la programmation orientée objet tiré directement du monde qui nous entoure. Tout élément du monde réel hérite d'un autre en le spécialisant, ainsi en tant qu'être humain nous héritons de toutes les caractéristiques d'un mammifère, au même titre qu'une pomme hérite des caractéristiques du fruit.

L'héritage est utilisé lorsqu'une relation de type « est-un » est possible. Nous pouvons considérer bien qu'un administrateur *est un* type de joueur et possède toutes ses capacités, et d'autres qui font de lui un modérateur. Lorsque cette relation n'est pas vérifiée, l'héritage ne doit pas être considéré, dans ce cas nous préférons généralement la *composition*. Nous traiterons au cours du chapitre 10 intitulé *Héritage versus composition* les différences entre les deux approches, nous verrons que l'héritage peut s'avérer rigide et difficile à maintenir dans certaines situations.

Dans un contexte d'héritage, des classes filles héritent automatiquement des fonctionnalités des classes mères. Dans notre cas, nous allons créer une classe `Administrateur` héritant de toutes les fonctionnalités de la classe `Joueur`. La classe mère est généralement appelée *super-classe*, tandis que la classe fille est appelée *sous-classe*.

Nous allons définir une classe `Administrateur` à côté de la classe `Joueur`, afin de traduire l'héritage nous utilisons le mot clé `extends` :

```
package  
  
{  
  
    // l'héritage est traduit par le mot clé extends  
    public class Administrateur extends Joueur  
  
    {
```

```
        public function Administrateur ( )
        {
        }
    }
}
```

Puis nous instancions un objet administrateur :

```
| var monModo:Administrateur = new Administrateur();
```

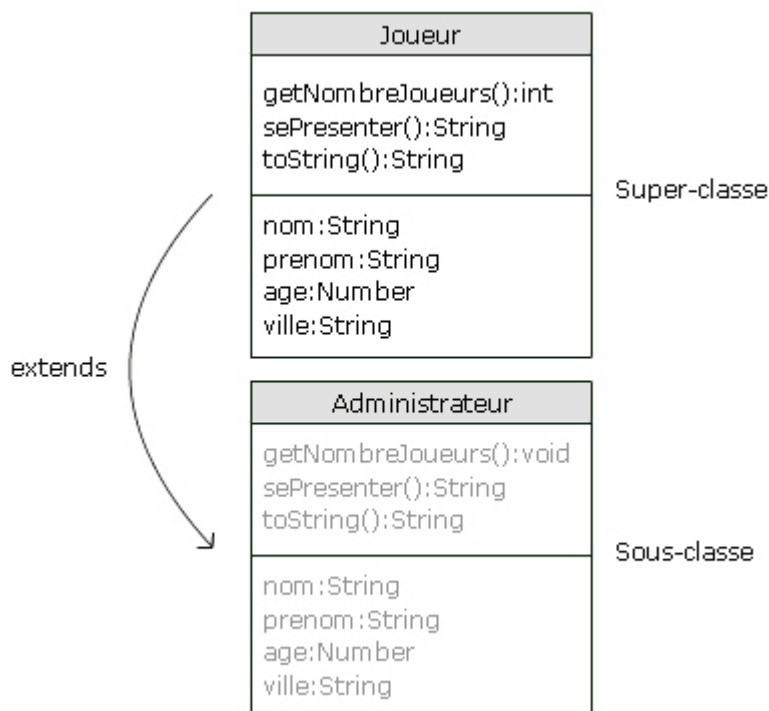
A la compilation, l'erreur suivante est générée :

```
| 1203: Aucun constructeur par défaut n'a été défini dans la classe de base
Joueur.
```

Lorsqu'une classe fille étend une classe mère, nous devons obligatoirement passer au constructeur parent, les paramètres nécessaires à l'initialisation de la classe mère, pour déclencher le constructeur parent nous utilisons le mot clé **super** :

```
package
{
    // l'héritage est traduit par le mot clé extends
    public class Administrateur extends Joueur
    {
        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )
        {
            super ( pPrenom, pNom, pAge, pVille );
        }
    }
}
```

L'intérêt de l'héritage réside dans la réutilisation du code. La figure 8-5 illustre le concept d'héritage de classes :



*Figure 8-5. Héritage de classes.*

Sans l'héritage nous aurions dû redéfinir l'ensemble des méthodes de la classe `Joueur` au sein de la classe `Administrateur`.

Grâce à l'héritage exprimé par le mot clé `extends`, la classe fille `Administrateur` hérite de toutes les fonctionnalités de la classe mère `Joueur` :

```

// nous créons un objet administrateur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// le modérateur possède toutes les capacités d'un joueur
// affiche : Je m'appelle Michael, j'ai 48 ans.
monModo.sePresenter();

// affiche : Jackson
trace( monModo.nom );

// affiche : Michael
trace( monModo.prenom );

// affiche : 48
trace( monModo.age );

// affiche : Los Angeles
trace( monModo.ville );
  
```

Il existe quelques exceptions liées à l'héritage, les méthodes et propriétés statiques ne sont pas héritées. La méthode statique

`getNombreJoueurs` définie au sein de la classe `Joueur` n'est pas disponible sur la classe `Administrateur` :

```
// instantiation d'un administrateur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// la méthode statique Joueur.getNombreJoueurs n'est pas héritée
monModo.getNombreJoueurs();
```

L'erreur à la compilation suivante est générée :

```
1061: Appel à la méthode getNombreJoueurs peut-être non définie, via la
référence de type static Administrateur.
```

Notons que l'héritage multiple n'est pas géré en ActionScript 3, une classe ne peut hériter de plusieurs classes directes.

L'héritage ne se limite pas aux classes personnalisées, nous verrons au cours du chapitre 9 intitulé *Etendre Flash* comment étendre des classes natives de Flash afin d'augmenter les capacités de classes telles `Array`, `BitmapData` ou `MovieClip` et bien d'autres.

### A retenir :

- L'héritage permet de réutiliser facilement les fonctionnalités d'une classe existante.
- La classe mère est appelée super-classe, la classe fille est appelée sous-classe. On parle alors de super-type et de sous-type.
- Afin d'hériter d'une classe nous utilisons le mot clé `extends`.

### Sous-types et super-type

Grâce à l'héritage, la classe `Administrateur` possède désormais deux types, `Joueur` considéré comme le super-type et `Administrateur` comme sous-type :

```
// la classe Administrateur est aussi de type Joueur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// un modérateur est un joueur
// affiche : true
trace( monModo is Joueur );

// un modérateur est aussi un administrateur
// affiche : true
trace( monModo is Administrateur );
```

Partout où le super-type est attendu nous pouvons passer une instance de sous-type, ainsi une variable de type `Joueur` peut stocker un objet de type `Administrateur` :

```
// la classe Administrateur est aussi de type Joueur
```

```
var monModo:Joueur = new Administrateur("Michael", "Jackson", 48, "Los Angeles");

// un modérateur est un joueur
// affiche : true
trace( monModo is Joueur );

// un modérateur est aussi un administrateur
// affiche : true
trace( monModo is Administrateur );
```

Tous les administrateurs sont des joueurs, le compilateur sait que toutes les fonctionnalités du type `Joueur` sont présentes au sein de la classe `Administrateur` et nous permet de compiler.

L'inverse n'est pas vrai, les joueurs ne sont pas forcément des administrateurs, il est donc impossible d'affecter un super-type à une variable de sous-type. Le code suivant ne peut être compilé :

```
// la classe Joueur n'est pas de type Administrateur
var monModo:Administrateur = new Joueur("Michael", "Jackson", 48, "Los Angeles");
```

L'erreur à la compilation suivante est générée :

```
1118: Contrainte implicite d'une valeur du type statique Joueur vers un type
peut-être sans rapport Administrateur.
```

Le compilateur ne peut nous garantir que les fonctionnalités de la classe `Administrateur` seront présentes sur l'objet `Joueur` et donc interdit la compilation.

Nous retrouvons le même concept en utilisant les classes graphiques natives comme `flash.display.Sprite` et `flash.display.MovieClip`, le code suivant peut être compilé sans problème :

```
// une instance de MovieClip est aussi de type Sprite
var monClip:Sprite = new MovieClip();
```

La classe `MovieClip` hérite de la classe `Sprite` ainsi une instance de `MovieClip` est aussi de type `Sprite`. A l'inverse, un `Sprite` n'est pas de type `MovieClip` :

```
// une instance de Sprite n'est pas de type MovieClip
var monClip:MovieClip = new Sprite();
```

Si nous testons le code précédent, l'erreur à la compilation suivante est générée :

```
1118: Contrainte implicite d'une valeur du type statique flash.display:Sprite
vers un type peut-être sans rapport flash.display:MovieClip.
```

La question que nous pouvons nous poser est la suivante, quel est l'intérêt de stocker un objet correspondant à un sous-type au sein

d'une variable de super-type ? Pourquoi ne pas simplement utiliser le type correspondant ?

Afin de bien comprendre ce concept, imaginons qu'une méthode nous renvoie des objets de différents types, prenons un cas très simple comme la méthode `getChildAt` de la classe `DisplayObjectContainer`.

Si nous regardons sa signature, nous voyons que celle-ci renvoie un objet de type `DisplayObject` :

```
| public function getChildAt(index:int):DisplayObject
```

En effet, la méthode `getChildAt` peut renvoyer toutes sortes d'objets graphiques tels `Shape`, `MovieClip`, `Sprite` ou bien `SimpleButton`. Tous ces types ont quelque chose qui les lie, le type `DisplayObject` est le type commun à tous ces objets graphiques, ainsi lorsque plusieurs types sont attendus, nous utilisons un type commun aux différentes classes.

Nous allons justement mettre cela en application au sein de notre classe `JoueurManager`, si nous regardons la signature de la méthode `ajouterJoueur` nous voyons que celle-ci accepte un paramètre de type `Joueur` :

```
| public function ajouteJoueur ( pJoueur:Joueur ):void  
| {  
|     pJoueur.sePresenter();  
|     tableauJoueurs.push ( pJoueur );  
| }
```

`Joueur` est le type commun aux deux classes `Joueur` et `Administrateur`, nous pouvons donc sans problème lui passer des instances de type `Administrateur` :

```
| // création des joueurs et du modérateur  
| var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");  
| var deuxiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48,  
| "Michigan");  
| var troisiemeJoueur:Joueur = new Joueur ("Lenny", "Williams", 50, "New  
| York");  
| var monModo:Administrateur = new Administrateur ("Stevie", "Wonder", 48, "Los  
| Angeles");  
|  
| // gestion des joueurs  
| var monManager:JoueurManager = new JoueurManager();  
|  
| // ajout des joueurs  
| /* affiche :  
| Je m'appelle Bobby, j'ai 66 ans.  
| Je m'appelle Michael, j'ai 48 ans.
```

```
Je m'appelle Bobby, j'ai 30 ans.  
Je m'appelle Stevie, j'ai 48 ans.  
Je suis modérateur  
*/  
monManager.ajouteJoueur ( premierJoueur );  
monManager.ajouteJoueur ( deuxiemeJoueur );  
monManager.ajouteJoueur ( troisiemeJoueur );  
monManager.ajouteJoueur ( monModo );
```

Si nous devons plus tard créer de nouveaux types de joueurs en étendant la classe `Joueur`, aucune modification ne sera nécessaire au sein de la fonction `ajouteJoueur`.

### A retenir :

- Grâce à l'héritage, une classe peut avoir plusieurs types.
- Partout où une classe mère est attendue nous pouvons utiliser une classe fille. C'est ce que nous appelons le *polymorphisme*.

### Spécialiser une classe

Pour le moment, la classe `Administrateur` possède les mêmes fonctionnalités que la classe `Joueur`. Il est inutile d'hériter simplement d'une classe mère sans ajouter de nouvelles fonctionnalités, en définissant de nouvelles méthodes au sein de la classe `Administrateur` nous spécialisons la classe `Joueur`.

Nous allons définir une nouvelle méthode appelée `kickJoueur` qui aura pour but de supprimer un joueur de la partie en cours :

```
package  
  
{  
  
    // l'héritage est traduit par le mot clé extends  
    public class Administrateur extends Joueur  
  
    {  
  
        public function Administrateur ( pPrenom:String, pNom:String,  
        pAge:int, pVille:String )  
  
        {  
  
            super ( pPrenom, pNom, pAge, pVille );  
  
        }  
  
        // méthode permettant de supprimer un joueur de la partie  
        public function kickJoueur ( pJoueur:Joueur ):void  
  
        {  
  
            trace ("Kick " + pJoueur );  
  
        }  
  
    }  
}
```

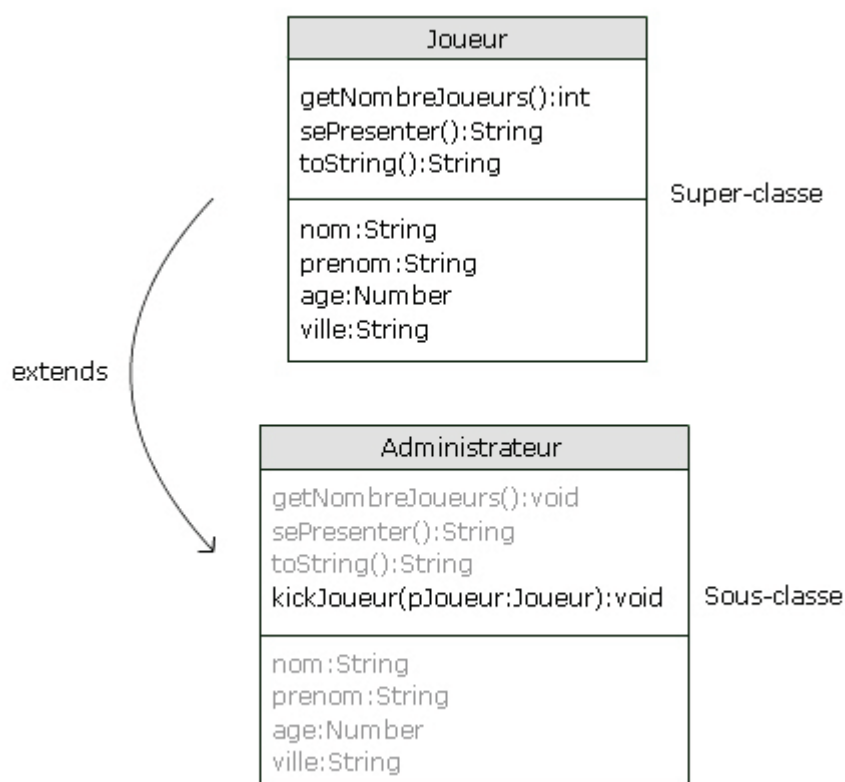
```

    }
}

```

Désormais la classe `Administrateur` possède une méthode `kickJoueur` en plus des fonctionnalités de la classe `Joueur`. Nous avons spécialisé la classe `Joueur` à travers la classe `Administrateur`.

La figure 8-6 illustre les deux classes :



*Figure 8-6. Spécialisation de la classe `Joueur`.*

Lorsque la méthode `kickJoueur` est exécutée, nous devons appeler la méthode `supprimeJoueur` de la classe `JoueurManager` car c'est celle qui centralise et gère les joueurs connectés.

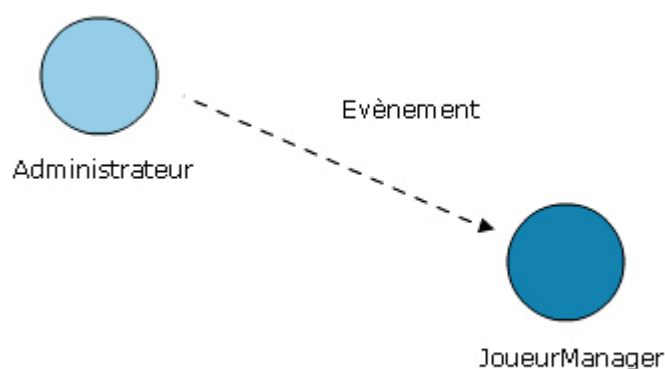
Chaque administrateur pourrait avoir une référence à la classe `JoueurManager` mais cela serait rigide, dans ce cas nous préférons une approche événementielle en utilisant dans nos classes personnalisées le modèle événementiel d'ActionScript 3 à l'aide de la classe `flash.events.EventDispatcher`.

La notion de diffusion d'événements personnalisés est traitée en détail lors du chapitre 12 intitulé *Diffusion d'événements personnalisés*. Une



fois achevé, n'hésitez pas à revenir sur cet exemple pour ajouter le code nécessaire. L'idée est que la classe `Administrateur` diffuse un événement indiquant à la classe `JoueurManager` de supprimer le joueur en question.

Nous retrouverons ici le concept de diffuseur écouteur traité dans les chapitres précédents. La figure 8-6 illustre le concept :



*Figure 8-6. L'objet `JoueurManager` est souscrit à un événement auprès d'un administrateur.*

Lorsqu'un administrateur diffuse l'événement approprié, l'objet `JoueurManager` supprime le joueur de la partie en cours. Nous allons nous intéresser maintenant à un processus appelé transtypage très utile dans un contexte d'héritage.

### A retenir :

- Il ne faut pas confondre héritage et spécialisation.
- Même si cela est déconseillé, une classe peut hériter d'une autre sans ajouter de nouvelles fonctionnalités.
- Lorsqu'une sous-classe ajoute de nouvelles fonctionnalités, on dit que celle-ci spécialise la super-classe.

## Le transtypage

Le transtypage est un processus très simple qui consiste à faire passer un objet pour un autre auprès du compilateur. Afin de comprendre le transtypage prenons la situation suivante, comme nous l'avons vu précédemment il peut arriver qu'une variable d'un type parent stocke des instances de types enfants.

Dans le code suivant nous stockons au sein d'une variable de type `Joueur` une instance de classe `Administrateur` :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monModo:Joueur = new Administrateur("Michael", "Jackson", 48, "Los
Angeles");

// la méthode kickJoueur n'existe pas pour le compilateur, car la classe
Joueur ne la définit pas
monModo.kickJoueur(premierJoueur);
```

Cela ne pose aucun problème, sauf lorsque nous tentons d'appeler la méthode `kickJoueur` sur la variable `monModo`, le compilateur nous génère l'erreur suivante :

```
1061: Appel à la méthode kickJoueur peut-être non définie, via la référence
de type static Joueur.
```

Pour le compilateur, la classe `Joueur` ne possède pas de méthode `kickJoueur`, la compilation est donc impossible. Pourtant nous savons qu'à l'exécution la variable `monModo` contiendra une instance de la classe `Administrateur` qui possède bien la méthode `kickJoueur`. Afin de faire taire le compilateur nous allons utiliser le transtypage en disant en quelque sorte au compilateur « Fais moi confiance, il y'a bien la méthode `kickJoueur` sur l'objet, laisse moi compiler ».

La syntaxe du transtypage est simple, nous précisons le type puis nous plaçons deux parenthèses comme pour une fonction traditionnelle :

```
| Type ( monObjet ) ;
```

Ainsi, nous transtypons la variable `monModo` vers le type `Administrateur` afin de pouvoir compiler :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monModo:Joueur = new Administrateur("Michael", "Jackson", 48, "Los
Angeles");

// affiche : Kick [Joueur prenom : Bobby, nom : Womack, age : 66, ville :
Detroit]
Administrateur ( monModo ).kickJoueur(premierJoueur);
```

A l'exécution, la méthode est bien exécutée. Là encore, nous pouvons nous demander dans quel cas concret nous devrions avoir recourt au transtypage ?

Nous allons définir une nouvelle méthode `jouerSon` sur la classe `Administrateur` qui sera automatiquement appelée au sein de la méthode `ajouteJoueur`. Lorsqu'un administrateur sera ajouté à

l'application un son sera joué afin de notifier de l'arrivée du modérateur :

```
package
{
    // l'héritage est traduit par le mot clé extends
    public class Administrateur extends Joueur
    {
        public function Administrateur ( pPrenom:String, pNom:String,
        pAge:int, pVille:String )
        {
            super ( pPrenom, pNom, pAge, pVille );
        }

        // méthode permettant de supprimer un joueur de la partie
        public function kickJoueur ( pJoueur:Joueur ):void
        {
            trace ("Kick " + pJoueur );
        }

        // méthode permettant de jouer un son
        public function jouerSon ( ):void
        {
            trace("Joue un son");
        }
    }
}
```

Au sein de la classe `JoueurManager` nous devons appeler la méthode `jouerSon` lorsqu'un objet de type `Administrateur` est passé, pour cela nous testons si le type correspond puis nous appelons la méthode voulue :

```
package
{
    public class JoueurManager
    {
        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array;

        public function JoueurManager ( )
        {
```

```
        tableauJoueurs = new Array();
    }

    public function ajouteJoueur ( pJoueur:Joueur ):void
    {
        pJoueur.sePresenter();

        if ( pJoueur is Administrateur ) pJoueur.jouerSon();

        tableauJoueurs.push ( pJoueur );
    }

    public function supprimeJoueur ( pJoueur:Joueur ):void
    {
        var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

        if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

        else throw new Error ("Joueur non présent !");
    }

    public function getJoueurs ( ):Array
    {
        return tableauJoueurs;
    }
}
}
```

Une fois nos classes modifiées, nous pouvons initialiser notre application :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var deuxiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48,
"Michigan");
var troisiemeJoueur:Joueur = new Joueur ("Lenny", "Williams", 50, "New
York");
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// gestion des joueurs
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs et administrateurs
monManager.ajouteJoueur ( premierJoueur );
monManager.ajouteJoueur ( deuxiemeJoueur );
monManager.ajouteJoueur ( troisiemeJoueur );
monManager.ajouteJoueur ( monModo );
```

A la compilation l'erreur suivante est générée :

```
1061: Appel à la méthode jouerSon peut-être non définie, via la référence de
type static Joueur.
```

Le compilateur se plaint car pour lui nous tentons d'appeler une méthode inexistante sur la classe `Joueur`. Nous savons qu'à l'exécution si la condition est validée, nous serons assurés que l'objet possède bien la méthode `jouerSon`, pour nous permettre de compiler nous transtypions vers le type `Administrateur`:

```
package
{

    public class JoueurManager
    {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array;

        public function JoueurManager ( )
        {

            tableauJoueurs = new Array();

        }

        public function ajouteJoueur ( pJoueur:Joueur ):void
        {

            pJoueur.sePresenter();

            if ( pJoueur is Administrateur ) Administrateur ( pJoueur
).jouerSon();

            tableauJoueurs.push ( pJoueur );

        }

        public function supprimeJoueur ( pJoueur:Joueur ):void
        {

            var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

            if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

            else throw new Error ("Joueur non présent !");

        }

        public function getJoueurs ( ):Array
        {

            return tableauJoueurs;

        }

    }

}
```

```
    }  
  }  
}
```

Nous pouvons initialiser notre application :

```
// création d'un joueur et d'un administrateur  
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");  
var deuxiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48,  
"Michigan");  
var troisiemeJoueur:Joueur = new Joueur ("Lenny", "Williams", 50, "New  
York");  
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,  
"Los Angeles");  
  
// gestion des joueurs  
var monManager:JoueurManager = new JoueurManager();  
  
// ajout des joueurs et administrateurs  
/* affiche :  
Je m'appelle Bobby, j'ai 66 ans.  
Je m'appelle Michael, j'ai 48 ans.  
Je m'appelle Lenny, j'ai 50 ans.  
Je m'appelle Michael, j'ai 48 ans.  
Joue un son  
*/  
monManager.ajouteJoueur ( premierJoueur );  
monManager.ajouteJoueur ( deuxiemeJoueur );  
monManager.ajouteJoueur ( troisiemeJoueur );  
monManager.ajouteJoueur ( monModo );
```

Nous reviendrons très souvent sur la notion de transtypage, couramment utilisée au sein la liste d’affichage.

## A retenir :

- Il ne faut pas confondre conversion et transtypage.
- Le transtypage ne convertit pas un objet en un autre, mais permet de faire passer un objet pour un autre.

## Surcharge

Le concept de surcharge (*override* en anglais) intervient lorsque nous avons besoin de modifier certaines fonctionnalités héritées. Imaginons qu’une méthode héritée ne soit pas assez complète, nous pouvons surcharger celle-ci dans la sous-classe afin d’intégrer notre nouvelle version.

Nous allons définir au sein de notre classe `Administrateur` une méthode `sePresenter`, afin de surcharger la version héritée, pour cela nous utilisons le mot clé `override` :

```
package
```

```
{  
  
    // l'héritage est traduit par le mot clé extends  
    public class Administrateur extends Joueur  
  
    {  
  
        public function Administrateur ( pPrenom:String, pNom:String,  
        pAge:int, pVille:String )  
  
        {  
  
            super ( pPrenom, pNom, pAge, pVille );  
  
        }  
  
        // méthode permettant à l'administrateur de se présenter  
        override public function sePresenter ( ):void  
  
        {  
  
            trace("Je suis modérateur");  
  
        }  
  
    }  
  
}
```

En appelant la méthode `sePresenter` sur notre instance de modérateur nous déclenchons la version redéfinie :

```
// nousinstancions un modérateur  
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,  
"Los Angeles");  
  
// nous lui demandons de se présenter  
// affiche : Je suis modérateur  
monModo.sePresenter();
```

En ActionScript 2, il n'existait pas de mot clé pour surcharger une méthode, le simple fait de définir une méthode du même nom au sein de la sous-classe surchargeait la méthode héritée. Grâce au mot clé `override` introduit par ActionScript 3 il est beaucoup plus simple pour un développeur de savoir si une méthode est une méthode surchargeante ou non.

Attention, la méthode surchargeante doit avoir le même nom et la même signature que la méthode surchargée, sinon la surcharge est dite *non compatible*. Dans l'exemple suivant nous retournons une valeur lorsque la méthode `sePresenter` est exécutée :

```
package  
  
{  
  
    // l'héritage est traduit par le mot clé extends  
    public class Administrateur extends Joueur
```

```
{
    public function Administrateur ( pPrenom:String, pNom:String,
    pAge:int, pVille:String )
    {
        super ( pPrenom, pNom, pAge, pVille );
    }

    // méthode permettant à l'administrateur de se présenter
    override public function sePresenter ( ):String
    {
        // déclenche la méthode surchargée
        super.sePresenter();

        return "Je suis modérateur";
    }

    // méthode permettant de supprimer un joueur de la partie
    public function kickJoueur ( pJoueur:Joueur ):void
    {
        trace ("Kick " + pJoueur );
    }

    // méthode permettant de jouer un son
    public function jouerSon ( ):void
    {
        trace("Joue un son");
    }
}
}
```

La méthode surchargeante n'a pas la même signature que la méthode surchargée, la compilation est impossible, si nous testons le code suivant :

```
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");
```

L'erreur suivante est générée à la compilation :

```
1023: override non compatible.
```

Dans un contexte de surcharge, la méthode surchargée n'est pas perdue. Si nous souhaitons au sein de notre méthode surchargeante déclencher la méthode surchargée, nous utilisons le mot clé **super** :

```
package
```



```
{  
  
    // l'héritage est traduit par le mot clé extends  
    public class Administrateur extends Joueur  
  
    {  
  
        public function Administrateur ( pPrenom:String, pNom:String,  
        pAge:int, pVille:String )  
  
        {  
  
            super ( pPrenom, pNom, pAge, pVille );  
  
        }  
  
        // méthode permettant au joueur de se présenter  
        override public function sePresenter ( ):void  
  
        {  
  
            // déclenche la méthode sePresenter surchargée  
            super.sePresenter();  
  
            trace("Je suis modérateur");  
  
        }  
  
    }  
  
}
```

Ainsi, lorsque la méthode `sePresenter` est déclenchée, celle-ci déclenche aussi la version surchargée :

```
// nousinstancions un modérateur  
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,  
"Los Angeles");  
  
// nous lui demandons de se présenter  
// affiche :  
/*  
Je m'appelle Michael, j'ai 48 ans.  
Je suis modérateur  
*/  
monModo.sePresenter();
```

Dans le code précédent nous augmentons les capacités de la méthode `sePresenter` en ajoutant le message « Je suis modérateur ». Grâce au mot clé `super` nous pouvons exécuter la méthode surchargée, celle-ci n'est pas perdue.

En surchargeant à l'aide d'une méthode vide, nous pouvons supprimer une fonctionnalité héritée :

```
package  
  
{
```

```
// l'héritage est traduit par le mot clé extends
public class Administrateur extends Joueur
{
    public function Administrateur ( pPrenom:String, pNom:String,
    pAge:int, pVille:String )
    {
        super ( pPrenom, pNom, pAge, pVille );
    }

    // méthode surchargeante vide
    override public function sePresenter ( ):void
    {
    }
}
}
```

Lorsque la méthode `sePresenter` est appelée sur une instance de la classe `Administrateur`, la version vide est exécutée. Celle-ci ne contenant aucune logique, nous avons annulé l'héritage de la méthode `sePresenter`.

### A retenir :

- La surcharge permet de redéfinir une fonctionnalité héritée.
- Afin que la surcharge soit possible, la méthode surchargée et surchargeante doivent avoir la même signature.
- Afin de surcharger une méthode nous utilisons le mot clé *override*.

# 9

## Etendre les classes natives

<b>INTERETS .....</b>	<b>1</b>
LE VIEIL AMI PROTOTYPE .....	2
<b>ETENDRE LES CLASSES NON GRAPHIQUES.....</b>	<b>5</b>
<b>ETENDRE LES CLASSES GRAPHIQUES.....</b>	<b>11</b>
ACCÉDER À L'OBJET STAGE DE MANIÈRE SÉCURISÉE .....	19
AJOUTER DES FONCTIONNALITÉS .....	23
RÉUTILISER LE CODE .....	47
CLASSE DYNAMIQUE .....	49
UN VRAI CONSTRUCTEUR .....	51
CRÉER DES BOUTONS DYNAMIQUES .....	52

### Intérêts

Nous avons découvert au cours du chapitre précédent les principaux concepts clé de la programmation orientée objet. Cependant, une des principales difficultés réside souvent dans la mise en application de ces notions dans un projet concret ActionScript.

La grande puissance de Flash réside dans sa capacité à lier graphisme et programmation. Nous allons profiter de cette force pour appliquer ce que nous avons appris au cours du chapitre précédent à travers différents cas pratiques.

La notion d'héritage ne se limite pas aux classes personnalisées et peut être appliquée à n'importe quelle classe de l'API du lecteur Flash. Il est par exemple possible d'étendre la classe native `MovieClip`, en définissant de nouvelles méthodes afin d'obtenir un `MovieClip` amélioré.

Le but de ce chapitre est d'apprendre à étendre les classes natives telles `MovieClip`, `Sprite`, `BitmapData` mais aussi des classes non graphiques comme `Array` afin d'augmenter les fonctionnalités offertes par celle-ci.

Attention, nous verrons que certaines classes natives sont considérées comme scellées et ne peuvent être étendues.

## Le vieil ami prototype

En ActionScript 1, les développeurs avaient pour habitude d'utiliser le `prototype` d'une classe afin d'augmenter ses capacités. En ActionScript 3, cette pratique fonctionne toujours mais n'est pas *officiellement* recommandée.

Dans le code suivant nous définissons une méthode `hello` sur le `prototype` de la classe `MovieClip` :

```
// ajout d'une méthode hello
MovieClip.prototype.hello = function ()
{
    // affiche : [object MovieClip]
    trace( this );
}

// création d'un clip
var monClip:MovieClip = new MovieClip();

// le clip possède automatiquement la méthode ajoutée au prototype
monClip.hello();
```

Automatiquement, le clip créé possède la méthode `hello`. Bien qu'efficace, cette technique pollue les autres instances de la même classe, car la méthode `hello` est alors disponible sur tous les `MovieClip` de l'animation.

Dans le code suivant nous appelons la méthode `hello` sur le scénario principal :

```
// ajout d'une méthode hello
MovieClip.prototype.hello = function ()
{
    // affiche : [object MainTimeline]
    trace( this );
}

// le scénario possède automatiquement la méthode ajoutée au prototype
this.hello();
```

Dans un concept d'héritage, l'idée est d'obtenir un nouveau type d'objet, doté de nouvelles fonctionnalités. En utilisant cette technique nous ne créons aucune nouvelle variété d'objet, nous ajoutons simplement des fonctionnalités à une classe existante.

Imaginons que nous devons créer une balle ayant la capacité de rebondir de manière élastique. Nous pourrions définir sur le `prototype` de la classe `MovieClip` toutes les méthodes de collision nécessaires à notre balle, mais serait-ce réellement optimisé ?

En utilisant cette approche, tous les `MovieClip` de notre application seraient dotés de capacités de rebond. Pourtant, seule la balle a véritablement besoin de telles capacités. Il serait plus intéressant d'étendre la classe `MovieClip` et de lier notre balle à la sous-classe.

Le prototypage possède en revanche un intérêt majeur lié à sa simplicité et son efficacité. Prenons le cas de la classe `DisplayObjectContainer`.

Comme nous l'avons vu lors du chapitre 4 intitulé *Liste d'affichage*, la classe `DisplayObjectContainer` ne définit pas de méthode permettant de supprimer la totalité des objets enfants.

A l'aide d'une méthode ajoutée au prototype de la classe `DisplayObjectContainer` nous pouvons ajouter très facilement cette fonctionnalité :

```
DisplayObjectContainer.prototype.supprimeEnfants = function ( )
{
    var nbEnfants:int = this.numChildren;

    while ( this.numChildren > 0 ) this.removeChildAt ( 0 );

    return nbEnfants;
}
```

Ainsi, toutes les instances de la classe `DisplayObjectContainer` disposent désormais d'une méthode `supprimeEnfants`. En plus de permettre la suppression de tous les enfants, celle-ci retourne le nombre d'enfants supprimés.

Si nous disposons plusieurs objets graphiques sur le scénario principal, nous pouvons les supprimer en appelant la méthode `supprimeEnfants` :

```
DisplayObjectContainer.prototype.supprimeEnfants = function ( )
{
```

```
        var nbEnfants:int = this.numChildren;

        while ( this.numChildren > 0 ) this.removeChildAt ( 0 );

        return nbEnfants;
    }

    // supprime tous les objets enfants du scénario principal
    this.supprimeEnfants();
```

De la même manière nous pouvons supprimer tous les objets enfants d'une instance de la classe `Sprite` :

```
// création d'un conteneur de type Sprite
var monSprite:Sprite = new Sprite ();

// création d'un objet Shape enfant
var maForme:Shape = new Shape();
maForme.graphics.lineStyle ( 1, 0x990000, 1 );
maForme.graphics.beginFill ( 0x990000, .2 );
maForme.graphics.drawCircle ( 50, 50, 50 );

monSprite.addChild( maForme );

addChild ( monSprite );

// suppression des objets enfants
var nbEnfantsSupprime:int = monSprite.supprimeEnfants();
```

En testant le code précédent, une erreur à la compilation est générée :

```
1061: Appel à la méthode supprimeEnfants peut-être non définie, via la
référence de type static flash.display.Sprite.
```

Le compilateur empêche la compilation car aucune méthode du nom de `supprimeEnfants` n'est trouvée. Afin de faire taire le compilateur nous pouvons exceptionnellement transtyper vers la classe dynamique `Object` non soumise à la vérification de type à la compilation :

```
// création d'un conteneur de type Sprite
var monSprite:Sprite = new Sprite ();

// création d'un objet Shape enfant
var maForme:Shape = new Shape();
maForme.graphics.lineStyle ( 1, 0x990000, 1 );
maForme.graphics.beginFill ( 0x990000, .2 );
maForme.graphics.drawCircle ( 50, 50, 50 );

monSprite.addChild( maForme );

addChild ( monSprite );

// suppression des objets enfants
var nbEnfantsSupprime:int = Object(monSprite).supprimeEnfants();

// affiche : 1
trace( nbEnfantsSupprime );
```

Nous reviendrons sur l'intérêt du prototypage au cours du chapitre 16 intitulé *Le texte*.

## A retenir

- L'utilisation du prototype est toujours possible en ActionScript 3.
- Son utilisation est *officiellement* déconseillée, mais offre une souplesse intéressante dans certains cas précis.
- Nous préférons dans la majorité des cas l'utilisation de sous-classes afin d'étendre les capacités.

## Etendre les classes non graphiques

D'autres classes natives peuvent aussi être étendues afin d'augmenter leurs capacités, c'est le cas de la classe `Array`, dont les différentes méthodes ne sont quelquefois pas suffisantes.

Ne vous est-il jamais arrivé de vouloir rapidement mélanger les données d'un tableau ?

En étendant la classe `Array` nous allons ajouter un ensemble de méthodes pratiques, qui seront disponibles pour toute instance de sous-classe. A côté d'un nouveau document Flash CS3 nous définissons une classe `MonTableau` au sein du paquetage `org.bytearray.ouils`.

Voici le code de la sous-classe `MonTableau` :

```
package org.bytearray.ouils
{
    dynamic public class MonTableau extends Array
    {
        public function MonTableau ( ...rest )
        {
        }
    }
}
```

La sous-classe `MonTableau` est une classe dynamique car l'accès aux données par l'écriture crochet requiert l'utilisation d'une classe dynamique.

Etendre la classe `Array` nécessite une petite astuce qui n'a pas été corrigée avec `ActionScript 3`. Au sein du constructeur de la sous-classe nous devons ajouter la ligne suivante :

```
package org.bytearray.ouutils
{
    dynamic public class MonTableau extends Array
    {
        public function MonTableau ( ...rest )
        {
            splice.apply(this, [0, 0].concat(rest));
        }
    }
}
```

Cette astuce permet d'initialiser correctement le tableau lorsque des paramètres sont passés au constructeur.

A ce stade, nous bénéficions d'une sous-classe opérationnelle, qui possède toutes les capacités d'un tableau standard :

```
// import de la classe MonTableau
import org.bytearray.ouutils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau (58, 48, 10);

// affiche : 48
trace( premierTableau[1] );

// ajout d'une valeur
premierTableau.push ( 25 );

// affiche : 4
trace( premierTableau.length );

// affiche : 25
trace( premierTableau.pop() );

// affiche : 3
trace( premierTableau.length );
```

Pour l'instant, l'utilisation de la sous-classe `MonTableau` n'est pas vraiment justifiée, celle-ci ne possède aucune fonctionnalité en plus de la classe `Array`.

Afin de copier un tableau, nous pourrions être tentés d'écrire le code suivant :

```
| // création d'un premier tableau
```

---



```
var monTableau:Array = new Array (58, 48, 10);

// création d'une copie ?
var maCopie:Array = monTableau;
```

Cette erreur fait référence à la notion de variables composites et primitives que nous avons traité lors du chapitre 2 intitulé *Langage et API du lecteur Flash*.

---

Souvenez-vous, lors de la copie de données composites, nous copions les variables par référence et non par valeur.

---

Afin de créer une vraie copie de tableau nous définissons une nouvelle méthode `copie` au sein de la classe `MonTableau` :

```
package org.bytearray.ouutils

{

    dynamic public class MonTableau extends Array

    {

        public function MonTableau ( ...rest )

        {

            splice.apply(this, [0, 0].concat(rest));

        }

        public function copie ( ):MonTableau

        {

            var maCopie:MonTableau = new MonTableau();
            var lng:int = length;

            for ( var i:int = 0; i< lng; i++ ) maCopie[i] = this[i];

            return maCopie;

        }

    }

}
```

En testant le code suivant, nous voyons qu'une réelle copie du tableau est retournée par la méthode `copie` :

```
// import de la classe MonTableau
import org.bytearray.ouutils.MonTableau;

// création d'un tableau de nombres
var tableau:MonTableau = new MonTableau (58, 48, 10);

// création d'une copie
```

```
var copieTableau:MonTableau = tableau.copie();  
  
// modification d'une valeur  
copieTableau[0] = 100;  
  
// affiche : 58,48,10 100,48,10  
trace ( tableau, copieTableau );
```

Nous aurions pu utiliser la méthode `Array.slice` mais celle-ci nous aurait renvoyé un tableau de type `Array` et non pas une nouvelle instance de `MonTableau`. En modifiant une valeur du tableau retourné nous voyons que les deux tableaux sont bien distincts.

Bien entendu, cette méthode ne fonctionne que pour la copie de tableaux contenant des données de types primitifs. Si nous souhaitons copier des tableaux contenant des données de types composites nous utiliserons la méthode `writeObject` de la classe `flash.utils.ByteArray`. Nous reviendrons sur cette fonctionnalité au cours du chapitre 20 intitulé *ByteArray*.

Nous allons maintenant ajouter une nouvelle méthode `melange` permettant de mélanger les données du tableau :

```
public function melange ( ):void  
{  
  
    var i:int = length;  
    var aleatoire:int;  
    var actuel:*;  
  
    while ( i-- )  
    {  
  
        aleatoire = Math.floor ( Math.random()*length );  
        actuel = this[i];  
        this[i] = this[aleatoire];  
        this[aleatoire] = actuel;  
  
    }  
  
}
```

Dans le code suivant nous mélangeons différentes valeurs, nous pourrions utiliser cette méthode dans un jeu. Nous pourrions imaginer une série de questions stockées dans un tableau, à chaque lancement le tableau est mélangé afin que les joueurs n'aient pas les questions dans le même ordre :

```
// import de la classe MonTableau  
import org.bytearray.ouutils.MonTableau;  
  
// création d'un tableau de nombres  
var tableau:MonTableau = new MonTableau(58, 48, 10);  
  
// mélange les valeurs
```

```
tableau.melange();  
  
// affiche : 85,12,58  
trace( tableau );
```

La méthode `egal` nous permet de tester si deux tableaux contiennent les mêmes valeurs :

```
public function egal ( pTableau:Array ):Boolean  
{  
  
    if ( this == pTableau ) return true;  
    var i:int = this.length;  
    if ( i != pTableau.length ) return false;  
    while( i-- )  
    {  
        if ( this[i] != pTableau[i] ) return false;  
    }  
    return true;  
  
}
```

Dans le code suivant nous comparons deux tableaux :

```
// import de la classe MonTableau  
import org.bytearray.ouutils.MonTableau;  
  
// création d'un tableau de nombres  
var premierTableau:MonTableau = new MonTableau(12, 58, 85);  
  
// création d'un autre tableau de nombres  
var secondTableau:MonTableau = new MonTableau(12, 58, 85);  
  
// affiche : true  
trace( premierTableau.egal ( secondTableau ) );
```

Lorsque le tableau passé contient les mêmes valeurs, la méthode `egal` renvoie `true`. A l'inverse si nous modifions les données du premier tableau, la méthode `egal` renvoie `false` :

```
// import de la classe MonTableau  
import org.bytearray.ouutils.MonTableau;  
  
// création d'un tableau de nombres  
var premierTableau:MonTableau = new MonTableau(100, 58, 85);  
  
// création d'un autre tableau de nombres  
var secondTableau:MonTableau = new MonTableau(12, 58, 85);  
  
// affiche : false  
trace( premierTableau.egal ( secondTableau ) );
```

Dans cet exemple nous ne prenons pas en charge les tableaux à plusieurs dimensions, différentes approches pourraient être utilisées, à vous de jouer !

Il serait pratique d'avoir une méthode qui se chargerait de calculer la moyenne des valeurs contenues dans le tableau. Avant de démarrer le

calcul nous devons nous assurer que le tableau ne contienne que des nombres.

Pour cela nous utilisons la nouvelle méthode `every` de la classe `Array`:

```
public function moyenne ( ):Number
{
    if ( ! every ( filtre ) ) throw new TypeError ( "Le tableau actuel ne
contient pas que des nombres" );

    var i:int = length;
    var somme:Number = 0;

    while ( i-- ) somme += this[i];

    return somme/length;
}

private function filtre ( pElement:*, pIndex:int, pTableau:Array ):Boolean
{
    return pElement is Number;
}
```

Lorsque la méthode `moyenne` est exécutée, celle-ci vérifie dans un premier temps si la totalité des données du tableau sont bien des nombres. Pour cela la méthode `filtre` est exécutée sur chaque élément du tableau, et renvoie `true` tant que l'élément parcouru est un nombre. Si ce n'est pas le cas, celle-ci renvoie `false` et nous levons une erreur de type `TypeError`. Si aucune erreur n'est levée, nous pouvons entamer le calcul de la moyenne.

Dans le code suivant, le calcul est possible :

```
// import de la classe MonTableau
import org.bytearray.utils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(100, 58, 85);

// affiche : 81
trace( premierTableau.moyenne() );
```

A l'inverse, si une valeur du tableau n'est pas un nombre :

```
// import de la classe MonTableau
import org.bytearray.utils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(this, 58, false);

trace( premierTableau.moyenne() );
```

Une erreur à l'exécution est levée :

```
TypeError: Le tableau actuel ne contient pas que des nombres
```

Afin de gérer l'erreur, nous pouvons placer l'appel de la méthode `moyenne` au sein d'un bloc `try catch` :

```
// import de la classe MonTableau
import org.bytearray.ouutils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(this, 58, false);

try
{
    trace( premierTableau.moyenne() );
} catch ( pError:Error )
{
    trace("une erreur de calcul est survenue !");
}
```

Nous pourrions ajouter autant de méthodes que nous souhaitons au sein de la classe `MonTableau`, libre à vous d'ajouter les fonctionnalités dont vous avez besoin. Cela nous permet de substituer la classe `MonTableau` à la classe `Array` afin de toujours avoir à disposition ces fonctionnalités.

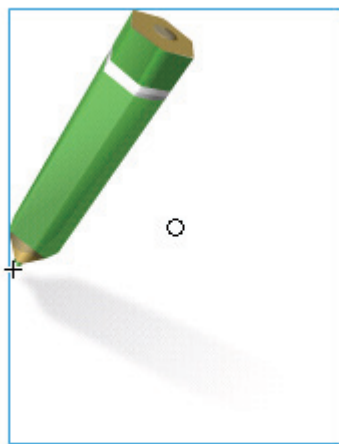
## A retenir

- Toutes les classes natives ne sont pas sous-classables.
- La composition peut être utilisée afin d'augmenter les capacités d'une classe qui n'est pas sous-classable.
- Etendre une classe native est un moyen élégant d'étendre les capacités d'ActionScript 3 ou du lecteur.

## Etendre les classes graphiques

L'extension de classes natives prend tout son sens dans le cas de sous-classes graphiques. Nous avons développé une application de dessin au cours du chapitre 7 intitulé *Intéractivité*, nous allons à présent l'enrichir en ajoutant un objet graphique tel un stylo.

Le graphisme a été réalisé sous 3D Studio Max puis ensuite exporté en image pour être utilisé dans Flash. Dans un nouveau document Flash CS3 nous importons le graphique puis nous le transformons en symbole clip.

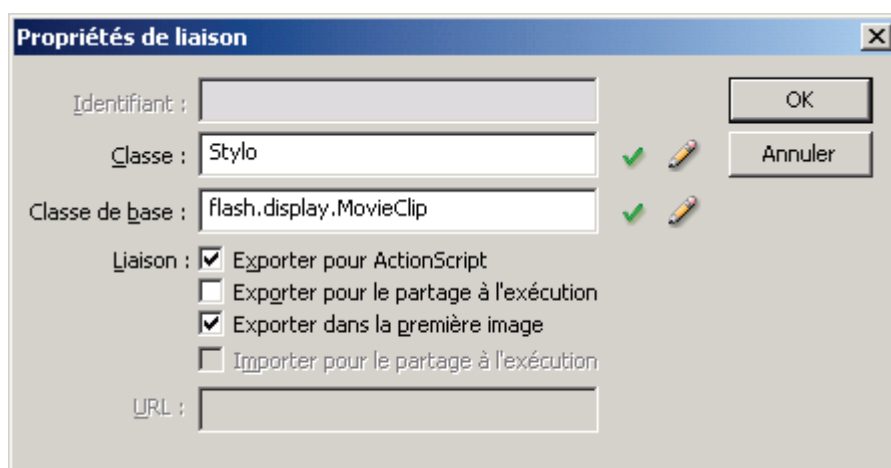


*Figure 9-1. Graphique du stylo converti en symbole clip.*

Attention à bien modifier le point d'enregistrement, de manière à ce que la mine du stylo soit en coordonnée 0,0.

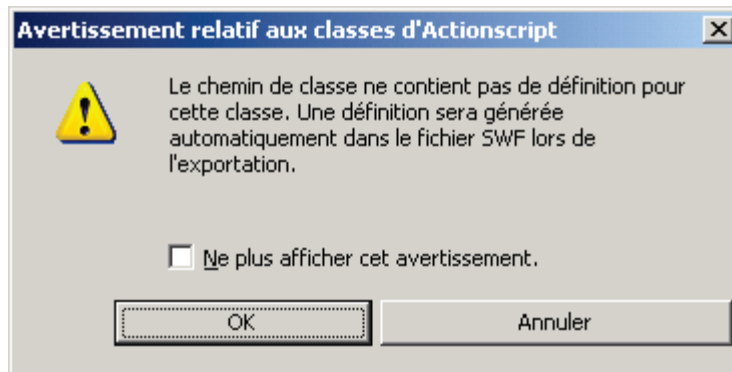
Ce clip va nous servir de graphisme afin de représenter le stylo de l'application. Nous devons pour cela le rendre disponible par programmation. Au sein du panneau *Propriétés de liaison* nous cochons la case *Exporter pour ActionScript* et renseignons *Stylo* comme nom de classe, nous laissons *flash.display.MovieClip* comme classe de base.

La classe *Stylo* est donc une sous-classe de *MovieClip* :



*Figure 9-2. Panneau propriétés de liaison.*

Lorsque nous cliquons sur le bouton OK, Flash tente de trouver une classe du même nom qui aurait pu être définie. Si il n'en trouve aucune, Flash affiche le message illustré en figure 9-3 :



*Figure 9-3. Génération automatique de classe.*

Comme nous l'avons vu au cours du chapitre 5 intitulé *Les symboles*, Flash génère automatiquement une classe interne utilisée pour instancier le symbole.

Dans notre cas, une classe *Stylo* est générée par Flash, afin de pouvoir instancier notre stylo puis l'afficher de la manière suivante :

```
// instantiation du stylo
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
addChild ( monStylo );
```

Flash génère automatiquement une classe *Stylo* qui hérite de la classe *MovieClip*, rappelez-vous que cette classe est inaccessible.

Si nous avions accès à celle-ci nous pourrions lire le code suivant :

```
package
{
    import flash.display.MovieClip;

    public class Stylo extends MovieClip
    {
        public function Stylo ()
        {

        }
    }
}
```

```
| }
```

La classe `Stylo` possède donc toutes les capacités d'un `MovieClip` :

```
// instantiation du stylo
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
addChild ( monStylo );

// la classe stylo possède toutes les capacités d'un clip
monStylo.stop();
monStylo.play();
monStylo.gotoAndStop (2);
monStylo.prevFrame ();
```

Cette génération automatique de classe s'avère très pratique lorsque nous ne souhaitons pas définir de classe manuellement pour chaque objet, Flash s'en charge et cela nous permet de gagner un temps précieux. En revanche Flash nous laisse aussi la possibilité de définir manuellement les sous classe graphiques, et d'y ajouter tout ce que nous souhaitons.

A coté de notre document Flash CS3 nous créons une classe `Stylo` héritant de `MovieClip` :

```
package
{
    import flash.display.MovieClip;

    public class Stylo extends MovieClip
    {
        public function Stylo ()
        {
            trace( this );
        }
    }
}
```

Nous sauvons la classe sous le nom `Stylo.as`, attention, le nom de la classe doit être le même que le fichier `.as`.

L'organisation de nos fichiers de travail doit être la suivante :

Nom	Taille	Type
Fl chap-9-dessin.fla	608 Ko	Document Flash
Stylo.as	8 Ko	Fichier Flash Action...

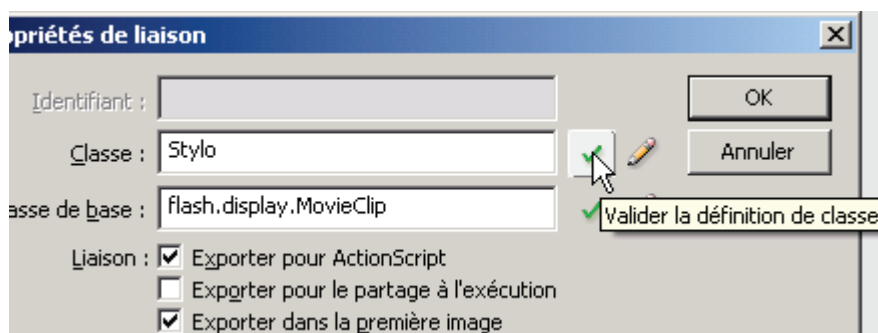


*Figure 9-4. Organisation des fichiers.*

Flash va désormais utiliser notre définition de classe et non celle générée automatiquement. Pour s'en assurer, nous utilisons le panneau *Propriétés de liaison*.

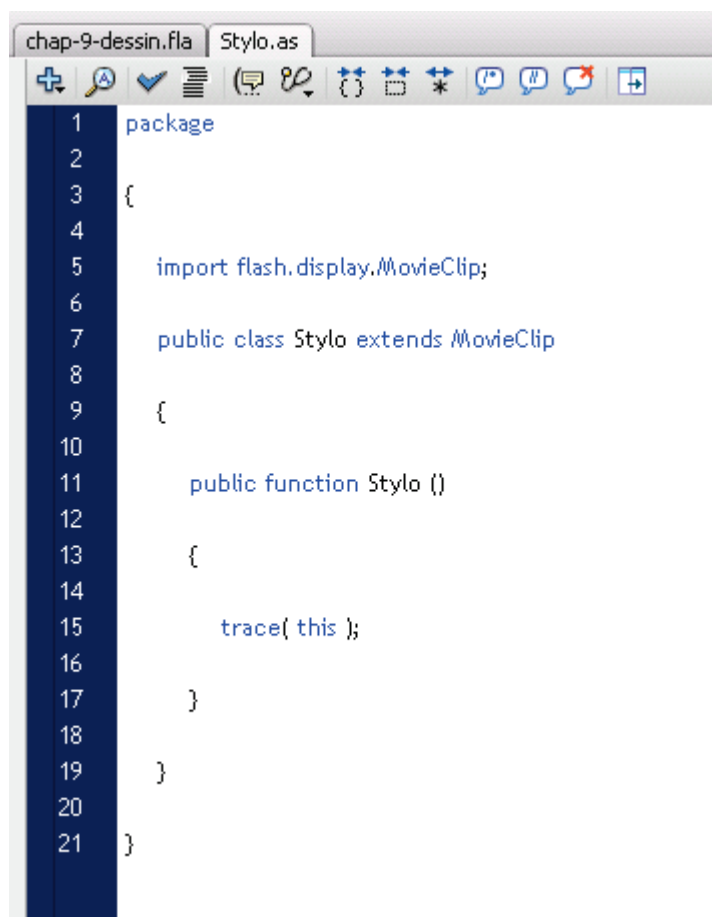
A droite du champ *Classe* sont situées deux icônes. En cliquant sur la première, nous pouvons valider la définition de la classe afin de vérifier que Flash utilise bien notre définition de classe.

La figure 9-5 illustre les deux icônes :



*Figure 9-5. Organisation des fichiers.*

Une fois la définition de classe validée, un message nous indiquant que la classe a bien été détectée s'affiche. Si nous cliquons sur l'icône d'édition représentant un stylo située à droite de l'icône de validation, la classe s'ouvre au sein de Flash afin d'être éditée :



*Figure 9-6. Edition de la classe au sein de Flash CS3.*

Le symbole est correctement lié à notre classe, nous pouvons dès à présent l’instancier :

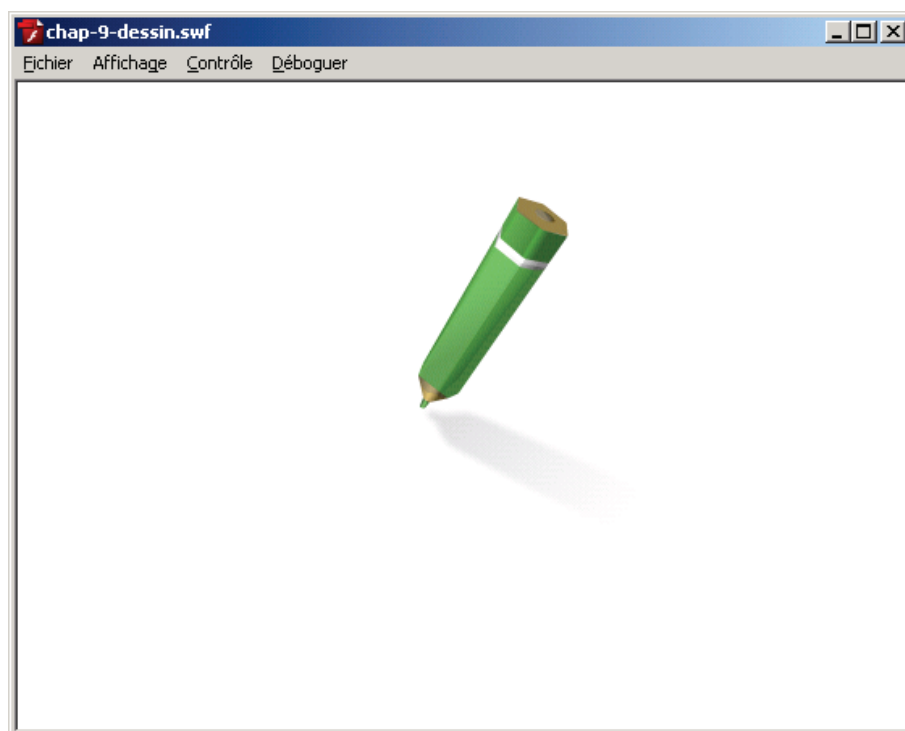
```
// affiche : [object Stylo]
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
addChild ( monStylo );

// positionnement en x et y
monStylo.x = 250;
monStylo.y = 200;
```

Lorsque le stylo est créé le constructeur est déclenché, l’instruction `trace` que nous avons placé affiche : `[object Stylo]`.

Notre symbole est affiché comme l’illustre la figure 9-7 :



*Figure 9-7. Symbole Stylo affiché.*

Nous n'avons défini pour le moment aucune nouvelle fonctionnalité au sein de la classe `Stylo`, celle-ci étend simplement la classe `flash.display.MovieClip`.

Toutes les fonctionnalités que nous ajouterons au sein de la classe `Stylo` seront automatiquement disponibles au sein du symbole, celui-ci est désormais *lié à la classe*.

En ajoutant une méthode `test` au sein de la classe `Stylo` :

```
package
{
    import flash.display.MovieClip;

    public class Stylo extends MovieClip
    {
        public function Stylo ()
        {
            trace( this );
        }

        public function test ( ):void
```

```
        {  
            trace("ma position dans l'axe des x est de : " + x );  
            trace("ma position dans l'axe des y est de : " + y );  
        }  
    }  
}
```

Celle-ci est automatiquement disponible auprès de l'instance :

```
// affiche : [object Stylo]  
var monStylo:Stylo = new Stylo();  
  
// ajout à la liste d'affichage  
addChild ( monStylo );  
  
// positionnement en x et y  
monStylo.x = 250;  
monStylo.y = 200;  
  
/* affiche :  
ma position dans l'axe des x est de : 250  
ma position dans l'axe des y est de : 200  
*/  
monStylo.test();
```

La première chose que notre stylo doit savoir faire est de suivre la souris. Nous n'allons pas utiliser l'instruction `startDrag` car nous verrons que nous devons travailler sur le mouvement du stylo, l'instruction `startDrag` ne nous le permettrait pas. Pour cela nous allons utiliser l'événement `MouseEvent.MOUSE_MOVE`.

Notre classe est une sous-classe de `MovieClip` et possède donc tous les événements interactifs nécessaires dont nous avons besoin, au sein du constructeur nous écoutons l'événement

`MouseEvent.MOUSE_MOVE` :

```
package  
{  
    import flash.display.MovieClip;  
    import flash.events.MouseEvent;  
  
    public class Stylo extends MovieClip  
    {  
        public function Stylo ()  
        {  
            trace( this );  
  
            addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );  
        }  
    }  
}
```

```
    }  
  
    private function bougeSouris ( pEvt:MouseEvent ):void  
    {  
  
        trace( pEvt );  
  
    }  
  
}  
  
}
```

Dans le code précédent la méthode `bougeSouris` est à l'écoute de l'événement `MouseEvent.MOUSE_MOVE`.

Pourquoi préférons-nous l'événement `MouseEvent.MOUSE_MOVE` à l'événement `Event.ENTER_FRAME` ?

Ce dernier se déclenche en continu indépendamment du mouvement de la souris. Ainsi, même si le stylo est immobile l'événement `Event.ENTER_FRAME` est diffusé. A terme, cela pourrait ralentir les performances de notre application.

Souvenez-vous que l'événement `MouseEvent.MOUSE_MOVE` n'est plus global comme en ActionScript 1 et 2, et n'est diffusé que lors du survol du stylo. Si nous avons besoin d'écouter le mouvement de la souris sur toute la scène nous devons accéder à l'objet `Stage`.

## A retenir

- Il est possible de lier un symbole existant à une sous-classe graphique définie manuellement.
- Le symbole hérite de toutes les fonctionnalités de la sous-classe.

## Accéder à l'objet Stage de manière sécurisée

Comme nous l'avons vu lors du chapitre 7 intitulé *Interactivité*, seul l'objet `Stage` permet une écoute globale de la souris ou du clavier. Nous devons donc modifier notre code afin d'écouter l'événement `MouseEvent.MOUSE_MOVE` auprès de l'objet `Stage` :

```
package  
  
{  
  
    import flash.display.MovieClip;  
    import flash.events.MouseEvent;  
  
    public class Stylo extends MovieClip  
    {
```

```
public function Stylo ()
{
    trace( this );

    stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

private function bougeSouris ( pEvt:MouseEvent ):void
{
    trace( pEvt );
}
}
```

En testant le code précédent, nous obtenons l'erreur suivante à l'exécution :

```
TypeError: Error #1009: Il est impossible d'accéder à la propriété ou à la
méthode d'une référence d'objet nul.
```

Souvenez-vous, lors du chapitre 4 intitulé *La liste d'affichage* nous avons vu que la propriété `stage` propre à tout objet de type `flash.display.DisplayObject` renvoyait `null` tant que l'objet graphique n'était pas ajouté à la liste d'affichage.

Ainsi, lorsque nous créons le symbole :

```
| var monStylo:Stylo = new Stylo();
```

Le constructeur est déclenché et nous tentons alors d'accéder à l'objet `Stage`. A ce moment là, notre symbole n'est pas encore présent au sein de la liste d'affichage, la propriété `stage` renvoie donc `null` et l'appel à la méthode `addEventListener` échoue.

Comment allons-nous procéder ?

Lors du chapitre 4 intitulé *La liste d'affichage* nous avons découvert deux événements importants liés à l'activation et à la désactivation des objets graphiques.

Voici un rappel de ces deux événements fondamentaux :

- `Event.ADDED_TO_STAGE` : cet événement est diffusé lorsque l'objet graphique est placé au sein d'un `DisplayObjectContainer` présent au sein de la liste d'affichage.
- `Event.REMOVED_FROM_STAGE` : cet événement est diffusé lorsque l'objet graphique est supprimé de la liste d'affichage.

Nous devons attendre que le symbole soit ajouté à la liste d’affichage pour pouvoir accéder à l’objet `Stage`. Le symbole va se souscrire lui-même auprès de l’événement `Event.ADDED_TO_STAGE` qu’il diffusera lorsqu’il sera ajouté à la liste d’affichage. Cela peut paraître étrange, mais dans ce cas l’objet s’écoute lui-même.

Nous modifions la classe `Stylo` afin d’intégrer ce mécanisme :

```
package

{

    import flash.display.MovieClip;
    import flash.events.Event;

    public class Stylo extends MovieClip

    {

        public function Stylo ()

        {

            trace( this );

            // le stylo écoute l'événement Event.ADDED_TO_STAGE, diffusé
            // lorsque celui-ci est ajouté à la liste d'affichage
            addEventListener ( Event.ADDED_TO_STAGE, activation );

        }

        private function activation ( pEvt:Event ):void

        {

            trace( pEvt );

        }

    }

}
```

Lors d’instanciation du symbole `Stylo`, le constructeur est déclenché :

```
// affiche : [object Stylo]
var monStylo:Stylo = new Stylo();
```

Lorsque l’instance est ajoutée à la liste d’affichage, l’événement `Event.ADDED_TO_STAGE` est diffusé :

```
// affiche : [object Stylo]
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
// affiche : [Event type="addedToStage" bubbles=false cancelable=false
// eventPhase=2]
addChild ( monStylo );
```

Nous définissons la méthode écouteur `activation` comme privée car nous n’y accéderons pas depuis l’extérieur. Gardez bien en tête de rendre privées toutes les méthodes et propriétés qui ne seront pas utilisées depuis l’extérieur de la classe.

L’événement `Event.ADDED_TO_STAGE` est diffusé, nous remarquons au passage que nous écoutons la phase cible et qu’il s’agit d’un événement qui ne participe pas à la phase de remontée.

Lorsque la méthode `activation` est déclenchée nous pouvons cibler en toute sécurité la propriété `stage` et ainsi écouter l’événement `MouseEvent.CLICK` :

```
package
{
    import flash.display.MovieClip;
    import flash.events.MouseEvent;
    import flash.events.Event;

    public class Stylo extends MovieClip
    {
        public function Stylo ()
        {
            // le stylo écoute l'événement Event.ADDED_TO_STAGE
            // diffusé lorsque celui-ci est ajouté à la liste d'affichage
            addEventListener ( Event.ADDED_TO_STAGE, activation );
        }

        private function activation ( pEvt:Event ):void
        {
            // écoute de l'événement MouseEvent.CLICK
            stage.addEventListener ( MouseEvent.CLICK, bougeSouris );
        }

        private function bougeSouris ( pEvt:MouseEvent ):void
        {
            trace( pEvt );
        }
    }
}
```

Si nous testons le code précédent, nous remarquons que la méthode `bougeSouris` est bien déclenchée lorsque la souris est déplacée sur la totalité de la scène.



## Ajouter des fonctionnalités

Nous allons à présent nous intéresser au mouvement du stylo en récupérant les coordonnées de la souris afin de le positionner.

Nous définissons deux propriétés `positionX` et `positionY` :

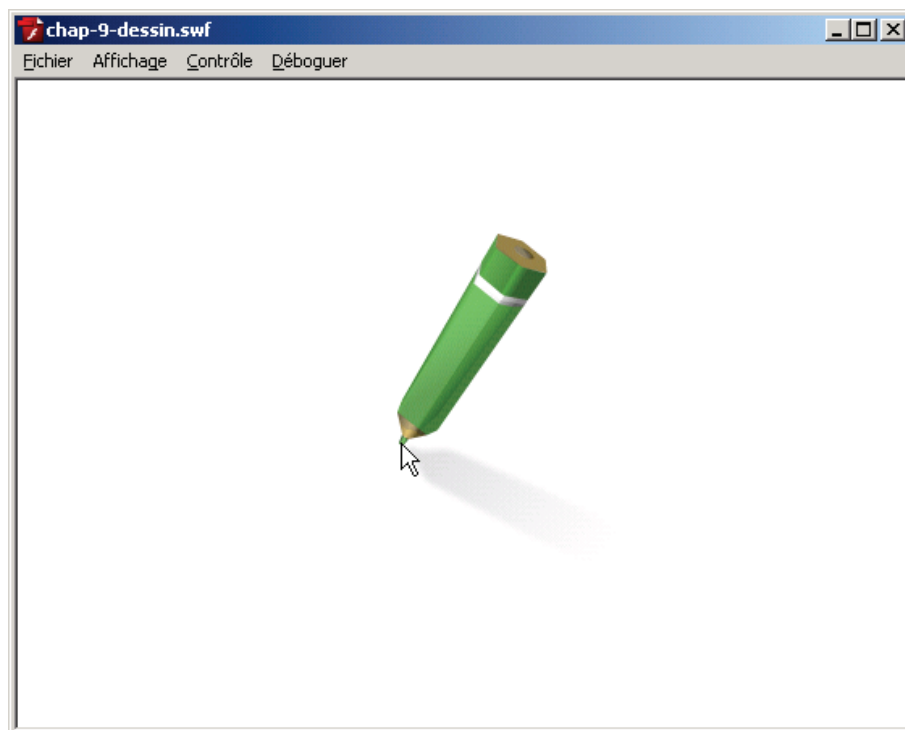
```
// position en cours de la souris
private var positionX:Number;
private var positionY:Number;
```

Celles-ci permettent de stocker la position de la souris :

```
private function bougeSouris ( pEvt:MouseEvent ):void
{
    // récupération des coordonnées de la souris en x et y
    positionX = pEvt.stageX;
    positionY = pEvt.stageY;

    // affectation de la position
    x = positionX;
    y = positionY;
}
```

Le stylo suit désormais la souris comme l'illustre la figure 9-8 :



*Figure 9-8. Stylo attaché au curseur.*

En testant l'application nous remarquons que le mouvement n'est pas totalement fluide, nous allons utiliser la méthode

`updateAfterEvent` que nous avons étudié au cours du chapitre 7 intitulé *Interactivité*.

Souvenez-vous, cette méthode de la classe `MouseEvent` nous permet de forcer le rafraîchissement du lecteur :

```
private function bougeSouris ( pEvt:MouseEvent ):void
{
    // récupération des coordonnées de la souris en x et y
    positionX = pEvt.stageX;
    positionY = pEvt.stageY;

    // affectation de la position
    x = positionX;
    y = positionY;

    // force le rafraîchissement
    pEvt.updateAfterEvent();
}
```

Le mouvement est maintenant fluide mais le curseur de la souris demeure affiché, nous le masquons à l'aide de la méthode statique `hide` de la classe `Mouse` lors de l'activation de l'objet graphique.

---

Veuillez à ne pas placer cet appel en continu au sein de la méthode `bougeSouris` cela serait redondant.

---

Nous importons la classe `Mouse` :

```
| import flash.ui.Mouse;
```

Puis nous modifions la méthode `activation` :

```
private function activation ( pEvt:Event ):void
{
    // cache le curseur
    Mouse.hide();

    // écoute de l'événement MouseEvent.MOUSE_MOVE
    stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}
```

A ce stade, nous avons étendu les capacités de la classe `MovieClip`, les symboles liés à la classe `Stylo` peuvent désormais suivre la souris. Nous pouvons à tout moment lier notre classe `Stylo` à n'importe quel autre symbole celui-ci bénéficiera aussitôt de toutes les fonctionnalités définies par celle-ci.

Nous avons pourtant oublié un élément essentiel, voyez-vous de quoi il s'agit ?

Nous devons gérer la désactivation de l'objet graphique afin de libérer les ressources. Si nous supprimons le stylo de la liste d'affichage, l'événement `MouseEvent.MOUSE_MOVE` est toujours diffusé, le curseur souris demeure masqué :

```
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
// affiche : [Event type="addedToStage" bubbles=false cancelable=false
eventPhase=2]
addChild ( monStylo );

// suppression du stylo, celui ci n'intègre aucune logique de désactivation
removeChild ( monStylo );
```

Pour gérer proprement cela nous écoutons l'événement `Event.REMOVED_FROM_STAGE` :

```
public function Stylo ()
{
    // le stylo écoute l'événement Event.ADDED_TO_STAGE
    // diffusé lorsque celui-ci est ajouté à la liste d'affichage
    addEventListener ( Event.ADDED_TO_STAGE, activation );

    // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
    // diffusé lorsque celui-ci est supprimé de la liste d'affichage
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
}
```

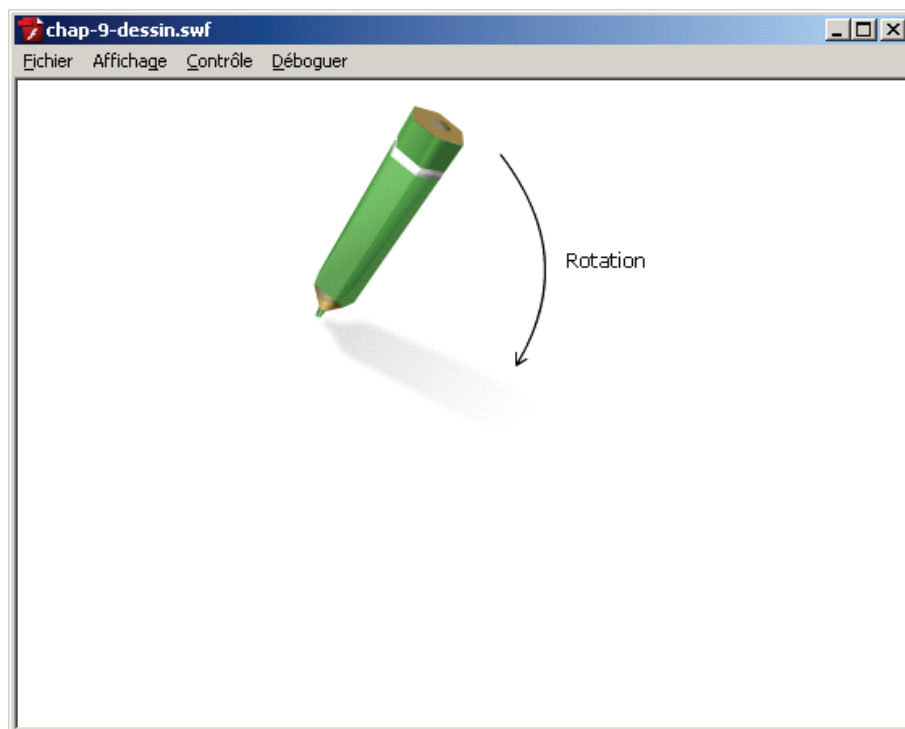
Puis nous ajoutons une méthode écouteur `desactivation` afin d'intégrer la logique de désactivation nécessaire :

```
private function desactivation ( pEvt:Event ):void
{
    // affiche le curseur
    Mouse.show();

    // arrête l'écoute de l'événement MouseEvent.MOUSE_MOVE
    stage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}
```

Il serait intéressant d'ajouter un effet de rotation au stylo lorsque celui-ci arrive en haut de la scène afin de simuler une rotation du poignet.

La figure 9-9 illustre l'idée :



*Figure 9-9. Rotation du stylo.*

En intégrant une simple condition nous allons pouvoir donner plus de réalisme au mouvement du stylo. Nous allons dans un premier temps définir la position dans l'axe des y à partir de laquelle le stylo commence à s'incliner, pour cela nous définissons une propriété constante car celle-ci ne changera pas à l'exécution :

```
// limite pour l'axe des y
private static const LIMIT_Y:int = 140;
```

Puis nous intégrons la condition au sein de la méthode `bougeSouris` :

```
private function bougeSouris ( pEvt:MouseEvent ):void
{
    // récupération des coordonnées de la souris en x et y
    positionX = pEvt.stageX;
    positionY = pEvt.stageY;

    // affectation de la position
    x = positionX;
    y = positionY;

    // si le stylo passe dans la zone supérieure alors nous inclinons le stylo
    if ( positionY < Stylo.LIMIT_Y )
    {
        rotation = ( Stylo.LIMIT_Y - positionY );
    }
}
```

```
    }  
  
    // force le rafraîchissement  
    pEvt.updateAfterEvent();  
  
}
```

A ce stade, voici le code complet de notre classe `Stylo` :

```
package  
  
{  
  
    import flash.display.MovieClip;  
    import flash.events.MouseEvent;  
    import flash.events.Event;  
    import flash.ui.Mouse;  
  
    public class Stylo extends MovieClip  
    {  
  
        // limite pour l'axe des y  
        private static const LIMIT_Y:int = 140;  
  
        // position en cours de la souris  
        private var positionX:Number;  
        private var positionY:Number;  
  
        public function Stylo ()  
        {  
  
            // le stylo écoute l'événement Event.ADDED_TO_STAGE  
            // diffusé lorsque celui-ci est ajouté à la liste d'affichage  
            addEventListener ( Event.ADDED_TO_STAGE, activation );  
  
            // le stylo écoute l'événement Event.REMOVED_FROM_STAGE  
            // diffusé lorsque celui-ci est supprimé de la liste d'affichage  
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );  
  
        }  
  
        private function activation ( pEvt:Event ):void  
        {  
  
            // cache le curseur  
            Mouse.hide();  
  
            // écoute de l'événement MouseEvent.MOUSE_MOVE  
            stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );  
  
        }  
  
        private function desactivation ( pEvt:Event ):void  
        {  
  
            // affiche le curseur  
            Mouse.show();  
  
        }  
  
    }  
}
```

```
        // arrête l'écoute de l'événement MouseEvent.MOUSE_MOVE
        stage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

    }

    private function bougeSouris ( pEvt:MouseEvent ):void
    {

        // récupération des coordonnées de la souris en x et y
        positionX = pEvt.stageX;
        positionY = pEvt.stageY;

        // affectation de la position
        x = positionX;
        y = positionY;

        // si le stylo passe dans la zone supérieure alors nous inclinons
le stylo    if ( positionY < Stylo.LIMIT_Y )
        {

            rotation = ( Stylo.LIMIT_Y - positionY );

        }

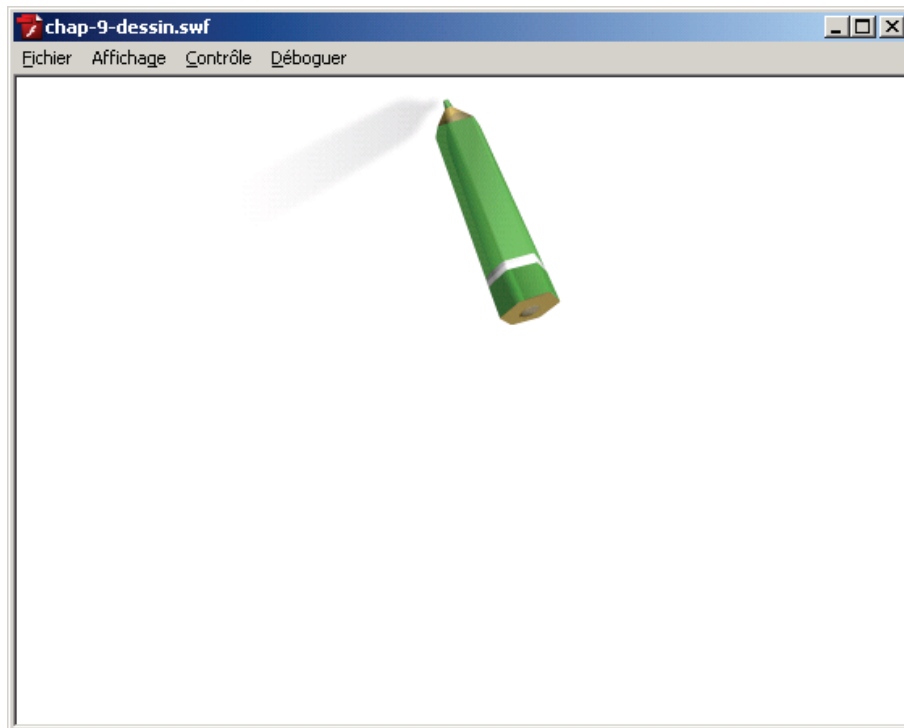
        // force le rafraîchissement
        pEvt.updateAfterEvent();

    }

}

}
```

Si nous testons l'application, lorsque nous arrivons en zone supérieure le stylo s'incline comme l'illustre la figure 9-10 :



*Figure 9-10. Inclinaison du stylo.*

Afin de rendre plus naturel le mouvement du stylo, nous ajoutons une formule d'inertie :

```
private function bougeSouris ( pEvt:MouseEvent ):void
{
    // récupération des coordonnées de la souris en x et y
    positionX = pEvt.stageX;
    positionY = pEvt.stageY;

    // affectation de la position
    x = positionX;
    y = positionY;

    // si le stylo passe dans la zone supérieure alors nous inclinons le stylo
    if ( positionY < Stylo.LIMIT_Y )
    {
        rotation -= ( rotation - ( Stylo.LIMIT_Y - positionY ) ) *.2;

        // sinon, le stylo reprend son inclinaison d'origine
    } else rotation -= ( rotation - 0 ) *.2;

    // force le rafraîchissement
    pEvt.updateAfterEvent();
}
```

Il est temps d'ajouter à présent la notion de dessin, cette partie a déjà été abordée lors du chapitre 7, nous allons donc réutiliser ce code dans notre application.

Si nous pensons en termes de séparation des tâches, il paraît logique qu'un stylo se charge de son mouvement et du dessin, et non pas à la création de la toile où dessiner. Nous allons donc définir une méthode `affecteToile` qui aura pour mission d'indiquer au stylo dans quel conteneur dessiner.

Nous définissons dans la classe une propriété permettant de référencer le conteneur :

```
| // stocke une référence au conteneur de dessin  
| private var conteneur:DisplayObjectContainer;
```

Puis nous importons la classe

```
flash.display.DisplayObjectContainer :
```

```
| import flash.display.DisplayObjectContainer;
```

Et ajoutons la méthode `affecteToile` :

```
| // méthode permettant de spécifier le conteneur du dessin  
| public function affecteToile ( pToile:DisplayObjectContainer ):void  
|  
| {  
|  
|     conteneur = pToile;  
|  
| }  
|
```

Tout type de conteneur peut être passé, à condition que celui-ci soit de type `DisplayObjectContainer` :

```
| // création du conteneur de tracés vectoriels  
| var toile:Sprite = new Sprite();  
|  
| // ajout du conteneur à la liste d'affichage  
| addChild ( toile );  
|  
| // création du symbole  
| var monStylo:Stylo = new Stylo();  
|  
| // affectation du conteneur de tracés  
| monStylo.affecteToile ( toile );  
|  
| // ajout du symbole à la liste d'affichage  
| addChild ( monStylo );  
|  
| // positionnement en x et y  
| monStylo.x = 250;  
| monStylo.y = 200;
```

Ainsi, différents types d'objets graphiques peuvent servir de toile. Souvenez-vous grâce à l'héritage un sous type peut être passé partout où un super-type est attendu.



En plus de réutiliser les fonctionnalités de dessin que nous avons développé durant le chapitre 7 intitulé *Interactivité* nous allons aussi réutiliser la notion d'historique combinés aux raccourcis clavier : CTRL+Z et CTRL+Y.

Nous définissons trois nouvelles propriétés permettant de stocker les formes créées et supprimées, puis le tracé en cours :

```
// tableaux référençant les formes tracées et supprimées
private var tableauTraces:Array = new Array();
private var tableauAncienTraces:Array = new Array();

// référence le tracé en cours
private var monNouveauTrace:Shape;
```

Attention, la classe `flash.display.Shape` doit être importée :

```
import flash.display.Shape;
```

Au sein de la méthode `activation`, nous devons tout d'abord écouter le clic souris, afin de savoir quand est-ce que l'utilisateur souhaite commencer à dessiner :

```
private function activation ( pEvt:Event ):void
{
    // cache le curseur
    Mouse.hide();

    // écoute des différents événements
    stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
    stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
}
```

Nous ajoutons la méthode écouteur `clicSouris`, qui se charge de créer les objets et d'initialiser le style du tracé :

```
private function clicSouris ( pEvt:MouseEvent ):void
{
    if ( conteneur == null ) throw new Error ( "Veuillez appeler au préalable la méthode affecteToile()" );

    // récupération des coordonnées de la souris en x et y
    positionX = pEvt.stageX;
    positionY = pEvt.stageY;

    // un nouvel objet Shape est créé pour chaque tracé
    monNouveauTrace = new Shape();

    // nous ajoutons le conteneur de tracé au conteneur principal
    conteneur.addChild ( monNouveauTrace );

    // puis nous référençons le tracé au sein du tableau
    // référençant les tracés affichés
    tableauTraces.push ( monNouveauTrace );
```

```

        // nous définissons un style de tracé
        monNouveauTrace.graphics.lineStyle ( 1, 0x990000, 1 );

        // la mine est déplacée à cette position
        // pour commencer à dessiner à partir de cette position
        monNouveauTrace.graphics.moveTo ( positionX, positionY );

        // si un nouveau tracé intervient alors que nous sommes
        // repartis en arrière nous repartons de cet état
        if ( tableauAncienTraces.length ) tableauAncienTraces = new Array;

        // écoute du mouvement de la souris
        stage.addEventListener ( MouseEvent.MOUSE_MOVE, dessine );
    }

```

Puis nous définissons la méthode `dessine` afin de gérer le tracé :

```

private function dessine ( pEvt:MouseEvent ):void
{
    if ( monNouveauTrace != null )
    {
        // la mine est déplacée à cette position
        // pour commencer à dessiner à partir de cette position
        monNouveauTrace.graphics.lineTo ( positionX, positionY );
    }
}

```

Afin d'arrêter de dessiner, nous écoutons le relâchement de la souris grâce à l'événement `MouseEvent.MOUSE_UP` :

```

private function activation ( pEvt:Event ):void
{
    // cache le curseur
    Mouse.hide();

    // écoute des différents événements
    stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
    stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
    stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );
}

```

La méthode écouteur `relacheSouris` est déclenchée lors du relâchement de la souris et interrompt l'écoute de l'événement `MouseEvent.MOUSE_MOVE` :

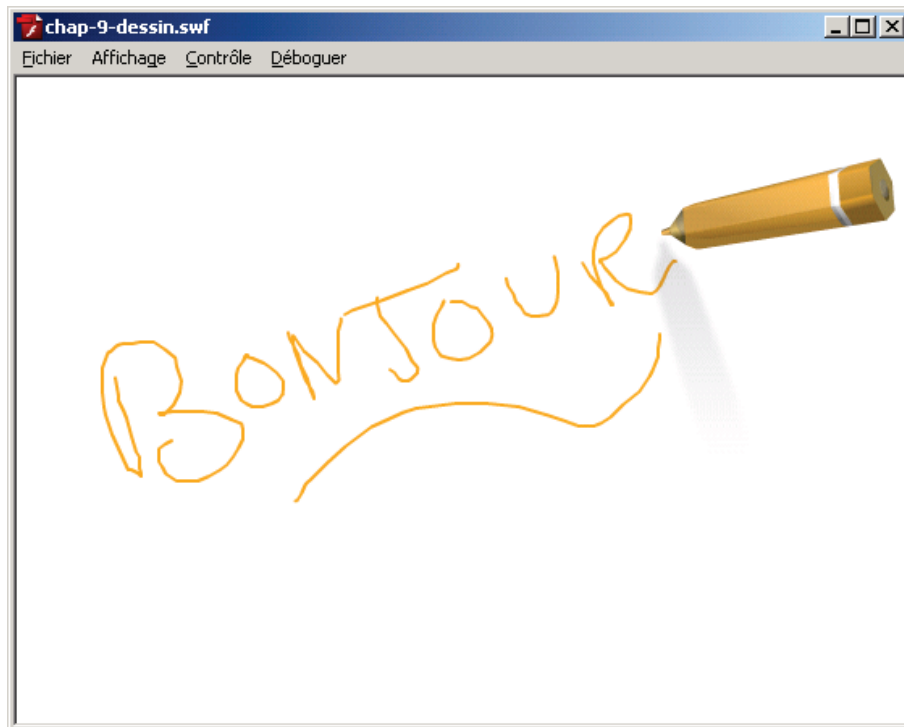
```

private function relacheSouris ( pEvt:MouseEvent ):void
{
    stage.removeEventListener ( MouseEvent.MOUSE_MOVE, dessine );
}

```

```
}
```

Si nous testons notre application, tout fonctionne correctement, nous pouvons à présent dessiner, lorsque la souris est relâchée, le tracé est stoppé.



*Figure 9-11. Application de dessin.*

Notre application est bientôt terminée, nous devons ajouter la notion d'historique pour cela nous importons la classe `KeyboardEvent` :

```
import flash.events.KeyboardEvent;
```

Puis nous ajoutons l'écoute du clavier au sein de la méthode `activation` :

```
private function activation ( pEvt:Event ):void
{
    // cache le curseur
    Mouse.hide();

    // écoute de l'événement MouseEvent.MOUSE_MOVE
    stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
    stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
    stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );
    stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );
}
```

La méthode écouteur `ecouteClavier` se charge de gérer l'historique, nous importons la classe `Keyboard` :

```
| import flash.ui.Keyboard;
```

Et définissons deux propriétés constantes stockant le code de chaque touche :

```
| // code des touches Y et Z
| private static const codeToucheY:int = 89;
| private static const codeToucheZ:int = 90;
```

Puis nous ajoutons le code déjà développé durant le chapitre 7 pour la précédente application de dessin :

```
private function ecouteClavier ( pEvt:KeyboardEvent ):void
{
    // si la barre espace est enfoncée
    if ( pEvt.keyCode == Keyboard.SPACE )
    {
        // nombre d'objets Shape contenant des tracés
        var lng:int = tableauTraces.length;

        // suppression des tracés de la liste d'affichage
        while ( lng-- ) conteneur.removeChild ( tableauTraces[lng] );

        // les tableaux d'historiques sont reinitialisés
        // les références supprimées
        tableauTraces = new Array();
        tableauAncienTraces = new Array();
        monNouveauTrace = null;
    }

    if ( pEvt.ctrlKey )
    {
        // si retour en arrière (CTRL+Z)
        if( pEvt.keyCode == Stylo.codeToucheZ && tableauTraces.length )
        {
            // nous supprimons le dernier tracé
            var aSupprimer:Shape = tableauTraces.pop();

            // nous stockons chaque tracé supprimé
            // dans le tableau spécifique
            tableauAncienTraces.push ( aSupprimer );

            // nous supprimons le tracé de la liste d'affichage
            conteneur.removeChild( aSupprimer );

            // si retour en avant (CTRL+Y)
        } else if ( pEvt.keyCode == Stylo.codeToucheY &&
tableauAncienTraces.length )
```

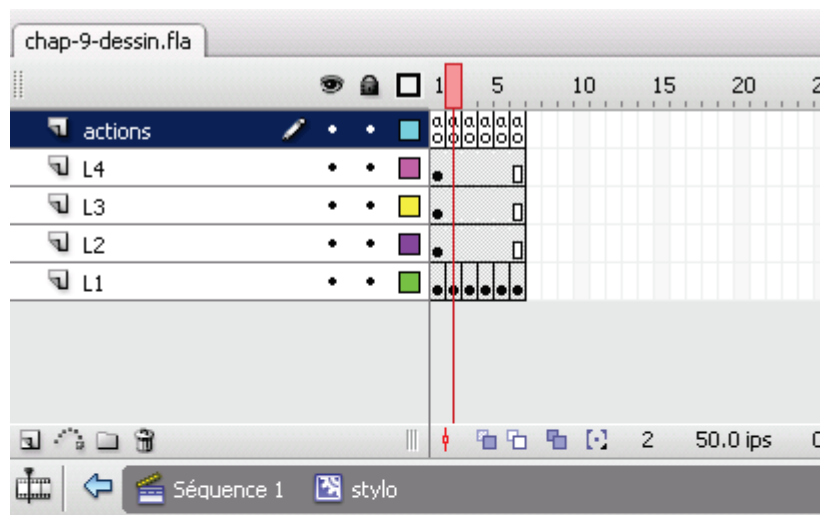
```
        {  
            // nous récupérons le dernier tracé ajouté  
            var aAfficher:Shape = tableauAncienTraces.pop();  
  
            // nous le remplaçons dans le tableau de  
            // tracés à l'affichage  
            tableauTraces.push ( aAfficher );  
  
            // puis nous l'affichons  
            conteneur.addChild ( aAfficher );  
        }  
    }  
}
```

Nous obtenons un symbole stylo intelligent doté de différentes fonctionnalités que nous avons ajoutées. Un des principaux avantages des sous-classes graphiques réside dans leur facilité de réutilisation.

Nous pouvons à tout moment lier un autre symbole à notre sous-classe graphique, ce dernier héritera automatiquement des fonctionnalités de la classe `Stylo` et deviendra opérationnel.

Avant cela, nous allons ajouter une autre fonctionnalité liée à la molette souris. Il serait élégant de pouvoir choisir la couleur du tracé à travers l'utilisation de la molette souris. Lorsque l'utilisateur s'en servira, la couleur du tracé et du stylo sera modifiée.

N'oublions pas qu'au sein de la sous-classe nous sommes sur le scénario du symbole. Nous modifions le symbole stylo en ajoutant plusieurs images clés :



*Figure 9-12. Ajout des différentes couleurs.*

Lorsque l'utilisateur fait usage de la molette, nous déplaçons la tête de lecture du scénario du symbole. Nous ajoutons deux nouvelles propriétés qui nous permettront de positionner la tête de lecture lorsque la souris sera glissée :

```
// position de la tête de lecture
private var image:int;
private var index:int;
```

Nous initialisons au sein du constructeur la propriété `index` qui sera plus tard incrémentée, nous l'initialisons donc à 0 :

```
public function Stylo ()
{
    // le stylo écoute l'événement Event.ADDED_TO_STAGE
    // diffusé lorsque celui-ci est ajouté à la liste d'affichage
    addEventListener ( Event.ADDED_TO_STAGE, activation );

    // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
    // diffusé lorsque celui-ci est supprimé de la liste d'affichage
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
```

```
// initialisation des propriétés utilisées pour
// le changement de couleurs
image = 0;
index = 1;

}
```

Au sein de la méthode `activation` nous ajoutons un écouteur de l'événement `MouseEvent.MOUSE_WHEEL` auprès de l'objet `Stage` :

```
private function activation ( pEvt:Event ):void
{
    // cache le curseur
    Mouse.hide();

    // écoute des différents événements
    stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
    stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
    stage.addEventListener ( MouseEvent.MOUSE_UP, lacheSouris );
    stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );
    stage.addEventListener ( MouseEvent.MOUSE_WHEEL, moletteSouris );
}
```

La méthode écouteur `moletteSouris` se charge de déplacer la tête de lecture sous forme de boucle, à l'aide de l'opérateur modulo `%`, la propriété `index` stocke l'index de la couleur sélectionnée :

```
// déplace la tête de lecture lors de l'utilisation de la molette
private function moletteSouris ( pEvt:MouseEvent ):void
{
    gotoAndStop ( index = (++image%totalFrames)+1 );
}
```

Si nous testons l'application, lorsque la molette de la souris est utilisée, le stylo change de couleur. Grâce au modulo, les couleurs défilent en boucle.

Il reste cependant à changer la couleur du tracé correspondant à la couleur du stylo choisie par l'utilisateur. Pour cela nous allons stocker toutes les couleurs disponibles dans un tableau au sein d'une propriété statique.

Pourquoi utiliser ici une propriété statique ?

Car les couleurs sont globales à toutes les occurrences du stylo qui pourraient être créées, ayant un sens global nous créons donc un tableau au sein d'une propriété `couleurs` statique :

```
// tableau contenant les couleurs disponibles
private static var couleurs:Array = [ 0x5BBA48, 0xEA312F, 0x00B7F1,
0xFFFF035, 0xD86EA3, 0xFBAE34 ];
```

Il ne nous reste plus qu'à modifier au sein de la méthode `clicSouris` la couleur du tracé, pour cela nous utilisons la propriété `index` qui représente l'index de la couleur au sein du tableau :

```
private function clicSouris ( pEvt:MouseEvent ):void
{
    if ( conteneur == null ) throw new Error ( "Veuillez appeler au
préalable la méthode affecteToile()" );

    // récupération des coordonnées de la souris en x et y
    positionX = pEvt.stageX;
    positionY = pEvt.stageY;

    // un nouvel objet Shape est créé pour chaque tracé
    monNouveauTrace = new Shape();

    // nous ajoutons le conteneur de tracé au conteneur principal
    conteneur.addChild ( monNouveauTrace );

    // puis nous référençons le tracé au sein du tableau
    // référençant les tracés affichés
    tableauTraces.push ( monNouveauTrace );

    // nous définissons un style de tracé
    monNouveauTrace.graphics.lineStyle ( 2, Stylo.couleurs[index-1], 1 );

    // la mine est déplacée à cette position
    // pour commencer à dessiner à partir de cette position
    monNouveauTrace.graphics.moveTo ( positionX, positionY );

    // si un nouveau tracé intervient alors que nous sommes
    // repartis en arrière nous repartons de cet état
    if ( tableauAncienTraces.length ) tableauAncienTraces = new Array;

    // écoute du mouvement de la souris
    stage.addEventListener ( MouseEvent.MOUSE_MOVE, dessine );
}
```

Lorsque la souris est enfoncée, nous pointons grâce à la propriété `index` au sein du tableau `couleurs` afin de choisir la couleur correspondante.

Nous ajoutons un petit détail final rendant notre classe plus souple, il serait intéressant de pouvoir en instanciant le stylo lui passer une vitesse permettant d'influencer sa vitesse de rotation.

Nous ajoutons une propriété friction de type `Number` :

```
// stocke la friction du stylo
private var friction:Number;
```

Puis nous modifions le constructeur afin d'accueillir la vitesse :

```
public function Stylo ( pFriction:Number=.1 )
{
```



```
// le stylo écoute l'événement Event.ADDED_TO_STAGE
// diffusé lorsque celui-ci est ajouté à la liste d'affichage
addEventListener ( Event.ADDED_TO_STAGE, activation );

// le stylo écoute l'événement Event.REMOVED_FROM_STAGE
// diffusé lorsque celui-ci est supprimé de la liste d'affichage
addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

// initialisation des propriétés utilisées pour
// le changement de couleurs
image = 0;
index = 1;

// affecte la friction
friction = pFriction;

}
```

Enfin, nous modifions la méthode `bougeSouris` afin d'utiliser la valeur passée, stockée au sein de la propriété `friction` :

```
private function bougeSouris ( pEvt:MouseEvent ):void
{
    // récupération des coordonnées de la souris en x et y
    positionX = pEvt.stageX;
    positionY = pEvt.stageY;

    // affectation de la position
    x = positionX;
    y = positionY;

    // si le stylo passe dans la zone supérieure
    // alors nous inclinons le stylo
    if ( positionY < Stylo.LIMIT_Y )
    {
        rotation -= ( rotation - ( Stylo.LIMIT_Y - positionY ) ) *
friction;

        // sinon, le stylo reprend son inclinaison d'origine
    } else rotation -= ( rotation - 0 ) * friction;

    // force le rafraîchissement
    pEvt.updateAfterEvent();
}
```

Souvenons-nous d'un concept important de la programmation orientée objet : l'encapsulation !

Dans notre classe `Stylo` les propriétés sont toutes privées, car il n'y aucune raison que celles-ci soient modifiables depuis l'extérieur. Bien entendu, il existe des situations dans lesquelles l'utilisation de propriétés privées n'a pas de sens.

Dans le cas d'une classe géométrique `Point`, il convient que les propriétés `x`, `y` et `z` soient publiques et accessibles. Il convient de cacher les propriétés qui ne sont pas utiles à l'utilisateur de la classe ou bien celles qui ont de fortes chances d'évoluer dans le temps.

---

Souvenez-vous, l'intérêt est de pouvoir changer l'implémentation sans altérer l'interface de programmation.

---

En utilisant des propriétés privées au sein de notre classe `Stylo`, nous garantissons qu'aucun code extérieur à la classe ne peut accéder et ainsi altérer le fonctionnement de l'application.

La friction qui est passée à l'initialisation d'un stylo doit obligatoirement être comprise entre 0 exclu et 1. Si ce n'est pas le cas, l'inclinaison du stylo échoue et le développeur assiste au dysfonctionnement de l'application.

Dans le code suivant le développeur ne connaît pas les valeurs possibles, et passe 10 comme vitesse d'inclinaison :

```
// création du symbole  
var monStylo:Stylo = new Stylo( 10 );
```

En testant l'application, le développeur se rend compte que le mécanisme d'inclinaison du stylo ne fonctionne plus. Cela est dû au fait que la valeur de friction exprime une force de frottement et doit être comprise entre 0 et 1.

N'oublions pas que nous sommes en train de développer un objet intelligent, autonome qui doit pouvoir évoluer dans différentes situations et pouvoir indiquer au développeur ce qui ne va pas.

Cela ne vous rappelle rien ?

Souvenez-vous d'un point essentiel que nous avons traité lors du précédent chapitre, *le contrôle d'affectation*.

Nous allons ajouter un test au sein du constructeur afin de vérifier si la valeur passée est acceptable, si ce n'est pas le cas nous passerons une valeur par défaut qui assurera le bon fonctionnement du stylo, à l'inverse si la valeur passée est incorrecte nous afficherons un message d'erreur.

Nous rajoutons la condition au sein du constructeur :

```
public function Stylo ( pFriction:Number=.1 )  
{  
  
    // le stylo écoute l'événement Event.ADDED_TO_STAGE
```

```
// diffusé lorsque celui-ci est ajouté à la liste d'affichage
addEventListener ( Event.ADDED_TO_STAGE, activation );

// le stylo écoute l'événement Event.REMOVED_FROM_STAGE
// diffusé lorsque celui-ci est supprimé de la liste d'affichage
addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

// initialisation des propriétés utilisées pour
// le changement de couleurs
image = 0;
index = 1;

// affecte la friction
if ( pFriction > 0 && pFriction <= 1 ) friction = pFriction;

else

{
    trace("Erreur : Friction non correcte, la valeur doit être
comprise entre 0 et 1");
    friction = .1;
}

}
```

Grâce à ce test, nous contrôlons l'affectation afin d'être sûr de la bonne exécution de l'application. Cette fois le développeur a passé une valeur supérieure à 1, l'affectation est contrôlée, un message d'information indique ce qui ne va pas :

```
/* affiche :
Erreur : Friction non correcte la valeur doit être comprise entre 0 et 1
*/
var monStylo:Stylo = new Stylo( 50 );
```

De cette manière, le développeur tiers possède toutes les informations pour s'assurer du bon fonctionnement du stylo. Elégant n'est ce pas ?

Il faut cependant prévoir la possibilité de changer la vitesse du mouvement une fois l'objet créé, pour cela nous allons définir une méthode appelée `affecteVitesse` qui s'occupera d'affecter la propriété `friction`.

Afin de bien encapsuler notre classe nous allons contrôler l'affectation des propriétés grâce à une méthode spécifique :

```
public function affecteVitesse ( pFriction:Number ):void

{

    // affecte la friction
    if ( pFriction > 0 && pFriction <= 1 ) friction = pFriction;

    else

    {
        trace("Erreur : Friction non correcte, la valeur doit être
comprise entre 0 et 1");
    }
}
```

```
        friction = .1;
    }
}
```

Nous intégrons le code défini précédemment au sein du constructeur, afin de contrôler l’affectation à la propriété `friction`. Nous pouvons donc remplacer le code du constructeur par un appel à la méthode `affecteVitesse` :

```
public function Stylo ( pFriction:Number=.1 )
{
    // le stylo écoute l'événement Event.ADDED_TO_STAGE
    // diffusé lorsque celui-ci est ajouté à la liste d'affichage
    addEventListener ( Event.ADDED_TO_STAGE, activation );

    // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
    // diffusé lorsque celui-ci est supprimé de la liste d'affichage
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

    // initialisation des propriétés utilisées pour
    // le changement de couleurs
    image = 0;
    index = 1;

    affecteVitesse ( pFriction );
}
```

En ajoutant une méthode `affecteVitesse`, l’affectation de la propriété `friction` est contrôlée.

Voici le code final de la classe `Stylo` :

```
package
{
    import flash.display.MovieClip;
    import flash.display.Shape;
    import flash.display.DisplayObjectContainer;
    import flash.events.MouseEvent;
    import flash.events.KeyboardEvent;
    import flash.events.Event;
    import flash.ui.Mouse;
    import flash.ui.Keyboard;

    public class Stylo extends MovieClip
    {
        // limite pour l'axe des y
        private static const LIMIT_Y:int = 140;

        // position en cours de la souris
        private var positionX:Number;
        private var positionY:Number;
```

```

// stocke une référence au conteneur de dessin
private var conteneur:DisplayObjectContainer;

// tableaux référençant les formes tracées et supprimées
private var tableauTraces:Array = new Array();
private var tableauAncienTraces:Array = new Array();

// référence le tracé en cours
private var monNouveauTrace:Shape;

// code des touches Y et Z
private static const codeToucheY:int = 89;
private static const codeToucheZ:int = 90;

// position de la tête de lecture
private var image:int;
private var index:int;

// tableau contenant les couleurs disponibles
private static var couleurs:Array = [ 0x5BBA48, 0xEA312F, 0x00B7F1,
0xFFFF035, 0xD86EA3, 0xFBAE34 ];

// stocke la friction du stylo
private var friction:Number;

public function Stylo ( pFriction:Number=.1 )
{
    // le stylo écoute l'événement Event.ADDED_TO_STAGE
    // diffusé lorsque celui-ci est ajouté à la liste d'affichage
    addEventListener ( Event.ADDED_TO_STAGE, activation );

    // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
    // diffusé lorsque celui-ci est supprimé de la liste d'affichage
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

    // initialisation des propriétés utilisées pour
    // le changement de couleurs
    image = 0;
    index = 1;

    // affectation contrôlée de la vitesse
    affecteVitesse ( pFriction );
}

private function activation ( pEvt:Event ):void
{
    // cache le curseur
    Mouse.hide();

    // écoute des différents événements
    stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
    stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
    stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );
    stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );
    stage.addEventListener ( MouseEvent.MOUSE_WHEEL, moletteSouris );
}

```

```

    }

    private function desactivation ( pEvt:Event ):void

    {

        // affiche le curseur
        Mouse.show();

        // arrête l'écoute des différents événements
        stage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
        stage.removeEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
        stage.removeEventListener ( MouseEvent.MOUSE_UP, relacheSouris );
        stage.removeEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier
    );
        stage.removeEventListener ( MouseEvent.MOUSE_WHEEL, moletteSouris
    );

    }

    // déplace la tête de lecture au scroll souris
    private function moletteSouris ( pEvt:MouseEvent ):void

    {

        gotoAndStop ( index = (++image%totalFrames)+1 );

    }

    private function ecouteClavier ( pEvt:KeyboardEvent ):void

    {

        // si la barre espace est enfoncée
        if ( pEvt.keyCode == Keyboard.SPACE )

        {

            // nombre d'objets Shape contenant des tracés
            var lng:int = tableauTraces.length;

            // suppression des tracés de la liste d'affichage
            while ( lng-- ) conteneur.removeChild ( tableauTraces[lng]
        );

            // les tableaux d'historiques sont reinitialisés
            // les références supprimées
            tableauTraces = new Array();
            tableauAncienTraces = new Array();
            monNouveauTrace = null;

        }

        if ( pEvt.ctrlKey )

        {

            // si retour en arrière (CTRL+Z)
            if( pEvt.keyCode == Stylo.codeToucheZ &&
            tableauTraces.length )

            {

```

```

        // nous supprimons le dernier tracé
        var aSupprimer:Shape = tableauTraces.pop();

        // nous stockons chaque tracé supprimé
        // dans le tableau spécifique
        tableauAncienTraces.push ( aSupprimer );

        // nous supprimons le tracé de la liste d'affichage
        conteneur.removeChild( aSupprimer );

        // si retour en avant (CTRL+Y)
        } else if ( pEvt.keyCode == Stylo.codeToucheY &&
tableauAncienTraces.length )

        {

            // nous récupérons le dernier tracé ajouté
            var aAfficher:Shape = tableauAncienTraces.pop();

            // nous le remplaçons dans le tableau de tracés à
l'affichage

            tableauTraces.push ( aAfficher );

            // puis nous l'affichons
            conteneur.addChild ( aAfficher );

        }

    }

    private function clicSouris ( pEvt:MouseEvent ):void

    {

        if ( conteneur == null ) throw new Error ( "Veuillez appeler au
préalable la méthode affecteToile()" );

        // récupération des coordonnées de la souris en x et y
        positionX = pEvt.stageX;
        positionY = pEvt.stageY;

        // un nouvel objet Shape est créé pour chaque tracé
        monNouveauTrace = new Shape();

        // nous ajoutons le conteneur de tracé au conteneur principal
        conteneur.addChild ( monNouveauTrace );

        // puis nous référençons le tracé au sein du tableau
        // référençant les tracés affichés
        tableauTraces.push ( monNouveauTrace );

        // nous définissons un style de tracé
        monNouveauTrace.graphics.lineStyle ( 2, Stylo.couleurs[index-1],
1 );

        // la mine est déplacée à cette position
        // pour commencer à dessiner à partir de cette position
        monNouveauTrace.graphics.moveTo ( positionX, positionY );
    }

```

```

        // si un nouveau tracé intervient alors que nous sommes
        // repartis en arrière nous repartons de cet état
        if ( tableauAncienTraces.length ) tableauAncienTraces = new
Array;

        // écoute du mouvement de la souris
        stage.addEventListener ( MouseEvent.MOUSE_MOVE, dessine );

    }

    private function bougeSouris ( pEvt:MouseEvent ):void
    {

        // récupération des coordonnées de la souris en x et y
        positionX = pEvt.stageX;
        positionY = pEvt.stageY;

        // affectation de la position
        x = positionX;
        y = positionY;

        // si le stylo passe dans la zone supérieure
        // alors nous inclinons le stylo
        if ( positionY < Stylo.LIMIT_Y )
        {

            rotation -= ( rotation - ( Stylo.LIMIT_Y - positionY ) ) *
friction;

            // sinon, le stylo reprend son inclinaison d'origine
        } else rotation -= ( rotation - 0 ) * friction;

        // force le rafraîchissement
        pEvt.updateAfterEvent();

    }

    private function relacheSouris ( pEvt:MouseEvent ):void
    {

        stage.removeEventListener ( MouseEvent.MOUSE_MOVE, dessine );

    }

    private function dessine ( pEvt:MouseEvent ):void
    {

        if ( monNouveauTrace != null )
        {

            // la mine est déplacée à cette position
            // pour commencer à dessiner à partir de cette position
            monNouveauTrace.graphics.lineTo ( positionX, positionY );

        }

    }

}

```



```
// méthode permettant de spécifier le conteneur du dessin
public function affecteToile ( pToile:DisplayObjectContainer ):void
{
    conteneur = pToile;
}

public function affecteVitesse ( pFriction:Number ):void
{
    // affecte la friction
    if ( pFriction > 0 && pFriction <= 1 ) friction = pFriction;
    else
    {
        trace("Erreur : Friction non correcte, la valeur doit être
comprise entre 0 et 1");
        friction = .1;
    }
}
}
```

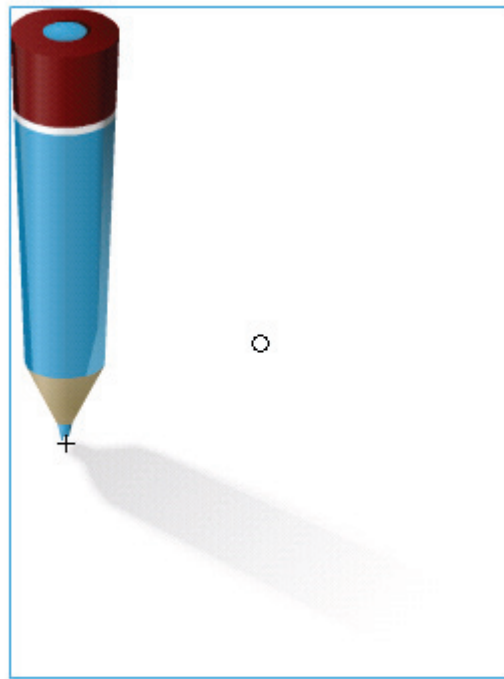
## A retenir

- Etendre les classes graphiques natives de Flash permet de créer des objets interactifs puissants et réutilisables.
- Au sein d'une sous-classe graphique, l'utilisation du mot clé `this` fait directement référence au scénario du symbole.

## Réutiliser le code

Notre application de dessin plaît beaucoup, une agence vient de nous contacter afin de décliner l'application pour un nouveau client. Grâce à notre structure actuelle, la déclinaison graphique ne va nécessiter aucune modification du code.

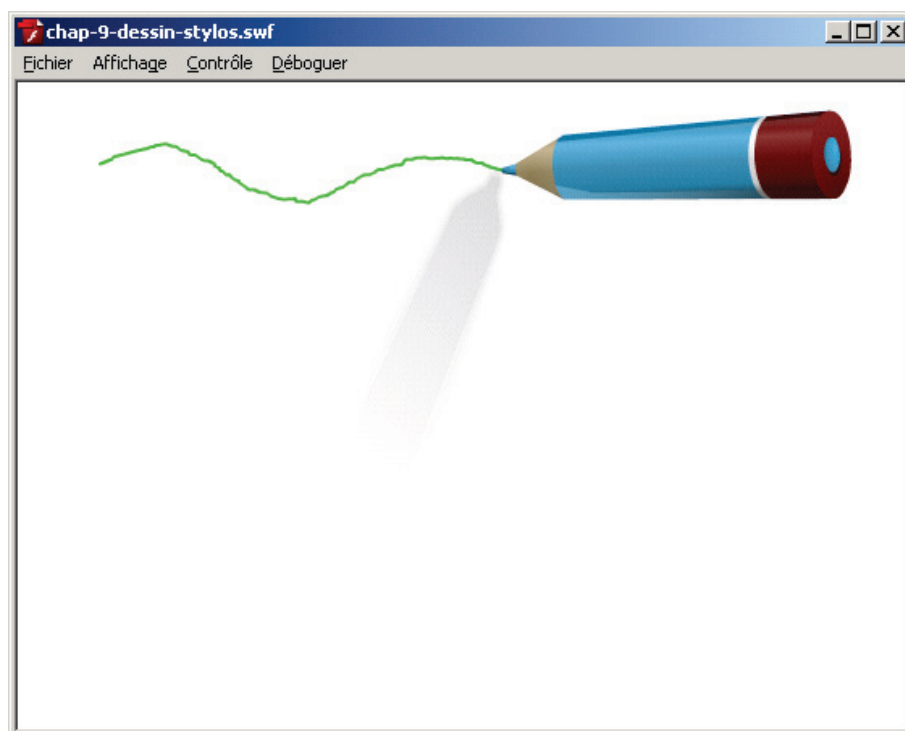
Dans un nouveau document Flash CS3 nous importons un nouveau graphisme relatif au stylo comme l'illustre la figure 9-13 :



*Figure 9-13. Nouveau graphisme.*

Au sein du panneau *Propriétés de liaison* nous spécifions la classe *Stylo* développée précédemment.

Voilà, nous pouvons compiler, la figure 9-14 illustre le résultat :



*Figure 9-14. Réutilisation du code de la classe `Stylo`.*

La déclinaison de notre projet a pris quelques secondes seulement, tout symbole de type `MovieClip` peut être lié à la classe `Stylo`, et bénéficier de toutes ses fonctionnalités et comportements.

## Classe dynamique

Comme nous l'avons vu lors des précédents chapitres, il existe en ActionScript 3 deux types de classes : dynamiques et non dynamiques.

Dans le cas de sous-classes graphiques, quelque soit la nature de super-classe, la sous-classe graphique est par défaut toujours non dynamique. Il est donc impossible à ce stade d'ajouter une propriété ou une méthode à l'exécution à une instance de la classe `Stylo` :

```
// création du symbole
var monStylo:Stylo = new Stylo( .1 );

// ajout d'une propriété à l'exécution
monStylo.maProp = 12;
```

Le code précédent génère l'erreur suivante à la compilation :

```
1119: Accès à la propriété maProp peut-être non définie, via la référence de
type static Stylo.
```

Ce comportement par défaut est de bon augure, car il est généralement déconseillé d'avoir recours à des classes dynamiques. En donnant la possibilité de modifier l'implémentation à l'exécution, le développeur

utilisant la classe doit parcourir le code d'un projet afin de découvrir les comportements qui peuvent être ajoutés à l'exécution. En lisant la classe, ce dernier n'est pas renseigné de toutes les capacités et caractéristiques de celle-ci.

Dans de rares situations, il peut toutefois être nécessaire de rendre la sous-classe graphique dynamique, pour cela nous ajoutons l'attribut `dynamic` devant le mot clé `class` :

```
| dynamic public class Stylo extends MovieClip
```

Une fois la classe modifiée, le code suivant fonctionne :

```
| // création du symbole  
| var monStylo:Stylo = new Stylo( .1 );  
  
| // ajout d'une propriété à l'exécution  
| monStylo.maProp = 12;  
  
| // affiche : 12  
| trace( monStylo.maProp );
```

Bien que l'attribut `dynamic` existe, seule la classe `MovieClip` en profite en ActionScript 3, toutes les autres classes ne permettent pas l'ajout de propriétés ou méthodes à l'exécution et sont dites non dynamiques.

Il est fortement déconseillé de s'appuyer sur des classes dynamiques dans vos projets. En rendant une classe dynamique aucune vérification de type n'est faite à la compilation, nous pouvons donc même si celle-ci nous renvoie `undefined`, accéder à des propriétés privées :

```
| // création du symbole  
| var monStylo:Stylo = new Stylo( .1 );  
  
| // aucune vérification de type à la compilation  
| // heureusement, la machine virtuelle (VM2) conserve les types  
| // à l'exécution et empêche son accès en renvoyant undefined  
| // affiche : undefined  
| trace( monStylo.nFriction );
```

La machine virtuelle 2 (VM2) conserve les types à l'exécution, ainsi lorsque nous accédons à une propriété privée au sein d'une classe dynamique, celle-ci renvoie `undefined`. Dans le cas d'une méthode privée, une erreur à l'exécution de type `TypeError` est levée.

---

## A retenir

---

- L'attribut `dynamic` permet de rendre une classe dynamique.
- Aucune vérification de type n'est faite à la compilation sur une classe dynamique.
- Il est déconseillé d'utiliser des classes dynamiques dans des projets ActionScript.
- Dans le cas de sous-classes graphiques, quelque soit la nature de super-classe, la sous-classe graphique est par défaut toujours non dynamique.

## Un vrai constructeur

En ActionScript 1 et 2, il était aussi possible d'étendre la classe `MovieClip`. Le principe était quasiment le même, un symbole était défini dans la librairie, puis une classe était liée au symbole.

Attention, il est important de noter qu'en ActionScript 2 la sous-classe graphique que nous pouvions définir était *liée au symbole*, cela signifie que seul l'appel de la méthode `attachMovie` instanciat la classe.

En ActionScript 3 grâce au nouveau modèle d'instanciation des objets graphiques, *c'est le symbole qui est lié à la classe*. Cela signifie que l'instanciation de la sous-classe entraîne l'affichage du symbole et non l'inverse.

Le seul moyen d'instancier notre sous-classe graphique en ActionScript 2 était d'appeler la méthode `attachMovie`. ActionScript 2, contrairement à ActionScript 3 souffrait d'un lourd héritage, ce processus d'instanciation des objets graphiques était archaïque et ne collait pas au modèle objet. En interne le lecteur déclenchait le constructeur de la sous-classe graphique. Il était impossible de passer des paramètres d'initialisation, seul l'objet d'initialisation (`initObject`) permettait de renseigner des propriétés avant même que le constructeur ne soit déclenché.

En instanciant la sous-classe, le symbole n'était pas affiché, le code suivant ne fonctionnait pas car le seul moyen d'afficher le clip pour le lecteur était la méthode `attachMovie` :

```
// la sous-classe était instanciée mais le symbole n'était pas affiché
var monSymbole:SousClasseMC = new SousClasseMC();
```

De plus, la liaison entre le symbole et la classe se faisait uniquement à travers le panneau *Propriétés de Liaison*, ainsi il était impossible de lier une classe à un objet graphique créé par programmation.

ActionScript 3 corrige cela. Dans l'exemple suivant nous allons créer une sous classe de `flash.display.Sprite` sans créer de symbole au sein de la bibliothèque, tout sera réalisé dynamiquement.

### A retenir

- En ActionScript 2, il était impossible de passer des paramètres au constructeur d'une sous classe graphique.
- Afin de palier à ce problème, nous utilisons l'objet d'initialisation disponible au sein de la méthode `attachMovie`.
- ActionScript 3 règle tout cela, grâce au nouveau modèle d'instanciation des objets graphiques.
- En ActionScript 2, la classe était liée au symbole. Seule la méthode `attachMovie` permettait d'instancier la sous-classe graphique.
- En ActionScript 3, le symbole est lié à la classe. L'instanciation de la sous-classe par le mot clé `new` entraîne la création du symbole.

### Créer des boutons dynamiques

Nous allons étendre la classe `Sprite` afin de créer des boutons dynamiques. Ces derniers nous permettront de créer un menu qui pourra être modifié plus tard afin de donner vie à d'autres types de menus.

Lors du chapitre 7 nous avons développé un menu composé de boutons de type `Sprite`. Tous les comportements étaient définis à l'extérieur de celui-ci, nous devons d'abord créer le symbole correspondant, puis au sein de la boucle ajouter les comportements boutons, les différents effets de survol, puis ajouter le texte. Il était donc impossible de recréer rapidement un bouton identique au sein d'une autre application.

En utilisant une approche orientée objet à l'aide d'une sous-classe graphique, nous allons pouvoir définir une classe `Bouton` qui pourra être réutilisée dans chaque application nécessitant un bouton fonctionnel.

Nous allons à présent organiser nos classes en les plaçant dans des paquetages spécifiques, au cours du chapitre précédent nous avons traité la notion de paquetages sans véritablement l'utiliser. Il est temps d'utiliser cette notion, cela va nous permettre d'organiser nos classes proprement et éviter dans certaines situations les conflits entre les classes.

Nous créons un nouveau document Flash CS3, à côté de celui-ci nous créons un répertoire `org` contenant un répertoire `bytearray`. Puis

au sein même de ce répertoire nous créons un autre répertoire appelé `ui`.

Au sein du répertoire `ui` nous créons une sous-classe de `Sprite` appelée `Bouton` contenant le code suivant :

```
package org.bytearray.ui

{
    import flash.display.Sprite;

    public class Bouton extends Sprite
    {

        public function Bouton ()

        {

            trace ( this );

        }

    }

}
```

Nous remarquons que le paquetage reflète l'emplacement de la classe au sein des répertoires, si le chemin n'est pas correct le compilateur ne trouve pas la classe et génère une erreur. Pour instancier la classe `Bouton` dans notre document Flash nous devons obligatoirement l'importer, car le compilateur recherche par défaut les classes situées à coté du fichier `.fla` mais ne parcourt pas tous les répertoires voisins afin de trouver la définition de classe.

Nous lui indiquons où elle se trouve à l'aide du mot clé `import` :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// affiche : [object Bouton]
var monBouton:Bouton = new Bouton();
```

Notre bouton est bien créé mais pour l'instant cela ne nous apporte pas beaucoup plus qu'une instanciation directe de la classe `Sprite`, nous allons tout de suite ajouter quelques fonctionnalités. Nous allons dessiner le bouton dynamiquement à l'aide de l'API de dessin. Grâce à l'héritage, la classe `Bouton` possède toutes les capacités d'un `Sprite` et hérite donc d'une propriété `graphics` propre à l'API de dessin.

Nous pourrions dessiner directement au sein de la classe `Bouton` mais nous préférons l'utilisation d'un objet `Shape` dédié à cela. Si nous devons plus tard ajouter un effet de survol nous déformerons celui-ci

afin de ne pas étirer l’enveloppe principale du bouton qui provoquerait un étirement global de tous les enfants du bouton.

Au sein du constructeur de la classe `Bouton` nous dessinons le bouton :

```
package org.bytearray.ui

{
    import flash.display.Shape;
    import flash.display.Sprite;

    public class Bouton extends Sprite
    {
        // référence le fond du bouton
        private var fondBouton:Shape;

        public function Bouton ()
        {
            // création du fond du bouton
            fondBouton = new Shape();

            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // dessine le bouton
            fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
            fondBouton.graphics.drawRect ( 0, 0, 40, 110 );
        }
    }
}
```

Nous choisissons une couleur aléatoire, puis nous dessinons une forme rectangulaire, plus tard nous pourrions choisir la couleur du bouton lors de sa création ou encore choisir sa dimension.

Il ne nous reste plus qu’à l’afficher :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

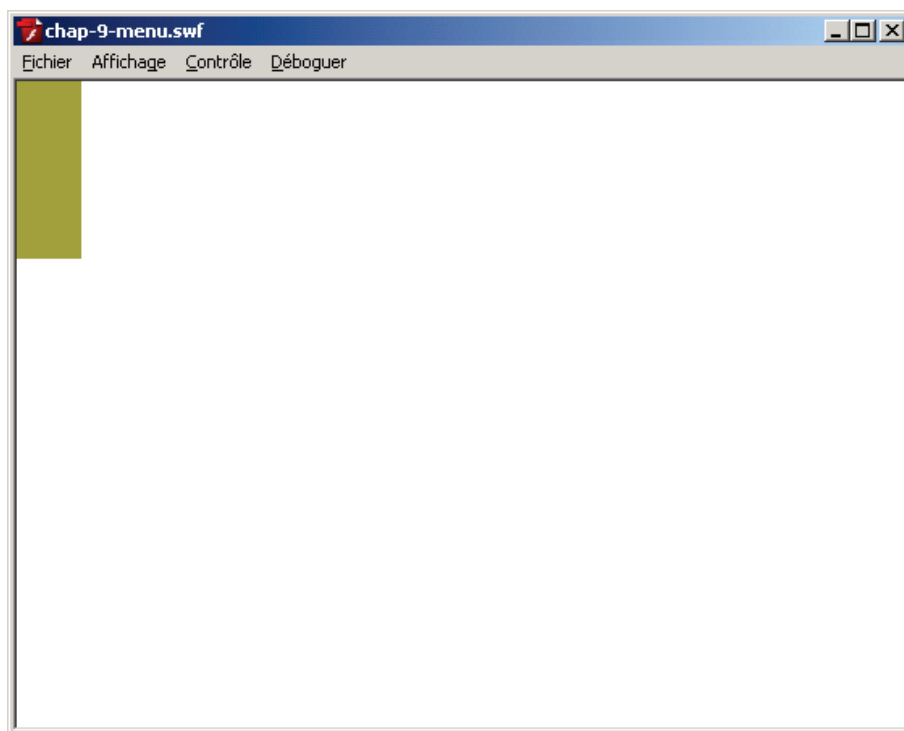
// instantiation
var monBouton:Bouton = new Bouton();

// ajout au sein du conteneur
conteneurMenu.addChild ( monBouton );

// affichage des boutons
addChild ( conteneurMenu );
```



Nous obtenons le résultat illustré en figure 9-15 :



*Figure 9-15. Instance de la classe Bouton.*

Afin de rendre notre bouton cliquable, nous devons activer une propriété ?

Vous souvenez-vous de laquelle ?

La propriété `buttonMode` permet d'activer le comportement bouton auprès de n'importe quel `DisplayObject` :

```
package org.bytearray.ui

{
    import flash.display.Shape;
    import flash.display.Sprite;

    public class Bouton extends Sprite
    {

        // stocke le fond du bouton
        private var fondBouton:Shape;

        public function Bouton ()
        {

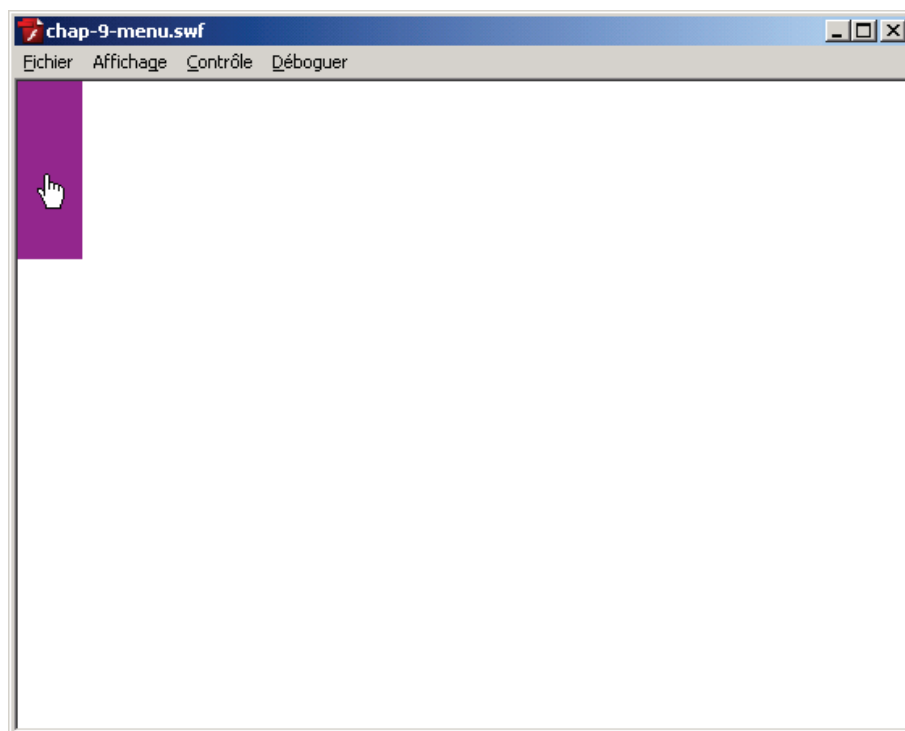
            // création du fond du bouton
            fondBouton = new Shape();
        }
    }
}
```

```
// ajout à la liste d'affichage
addChild ( fondBouton );

// dessine le bouton
fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

// activation du mode bouton
buttonMode = true;
    }
}
}
```

Le curseur main s’affiche alors au survol du bouton :



*Figure 9-16. Activation du mode bouton.*

Nous allons donner un peu de mouvement à ce dernier, en utilisant la classe **Tween** déjà abordée lors du chapitre 7.

Nous importons la classe **Tween** puis nous créons un objet du même type afin de gérer le survol :

```
package org.bytearray.ui
{
    import flash.display.Shape;
    import flash.display.Sprite;
    // import des classes Tween liées au mouvement
}
```

```
import fl.transitions.Tween;
import fl.transitions.easing.Bounce;

public class Bouton extends Sprite
{
    // stocke le fond du bouton
    private var fondBouton:Shape;
    // stocke l'objet Tween pour les différents états du bouton
    private var interpolation:Tween;

    public function Bouton ()
    {
        // création du fond du bouton
        fondBouton = new Shape();

        // ajout à la liste d'affichage
        addChild ( fondBouton );

        // dessine le bouton
        fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
        fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

        // activation du mode bouton
        buttonMode = true;

        // création de l'objet Tween
        interpolation = new Tween (fondBouton, "scaleX", Bounce.easeOut,
1, 1, 1, true );
    }
}
```

Nous demandons à l'objet **Tween** de s'occuper de la propriété **scaleX** pour donner un effet d'étirement à l'objet **Shape** servant de fond à notre bouton. Nous allons donner un effet de rebond exprimé par la classe **Bounce**.

Lorsque le bouton est survolé nous démarrons l'animation. Nous écoutons l'événement **MouseEvent.ROLL\_OVER** :

```
package org.bytearray.ui
{
    import flash.display.Shape;
    import flash.display.Sprite;
    // import des classes Tween liées au mouvement
    import fl.transitions.Tween;
    import fl.transitions.easing.Bounce;
    // import de la classe MouseEvent
    import flash.events.MouseEvent;

    public class Bouton extends Sprite
```

```
{

    // stocke le fond du bouton
    private var fondBouton:Shape;
    // stocke l'objet Tween pour les différents états du bouton
    private var interpolation:Tween;

    public function Bouton ()

    {

        // création du fond du bouton
        fondBouton = new Shape();

        // ajout à la liste d'affichage
        addChild ( fondBouton );

        // dessine le bouton
        fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
        fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

        // activation du mode bouton
        buttonMode = true;

        // création de l'objet Tween
        interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, 1, true );

        // écoute de l'événement MouseEvent.CLICK
        addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

    }

    // déclenché lors du survol du bouton
    private function survolSouris ( pEvt:MouseEvent ):void

    {

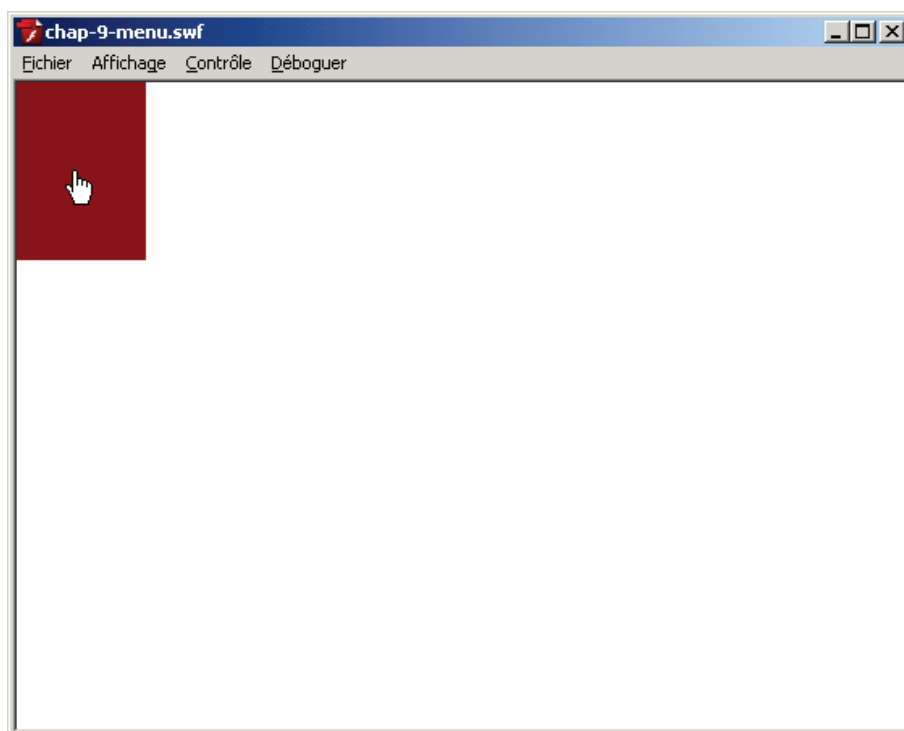
        // démarrage de l'animation
        interpolation.continueTo ( 2, 2 );

    }

}

}
```

Lors du survol, le bouton s'étire avec un effet de rebond, la figure 9-17 illustre l'effet :



*Figure 9-17. Etirement du bouton.*

En ajoutant d’autres boutons nous obtenons un premier menu :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

var lng:int = 5;

var monBouton:Bouton;

for (var i:int = 0; i< lng; i++ )
{

    // instanciation
    monBouton = new Bouton();

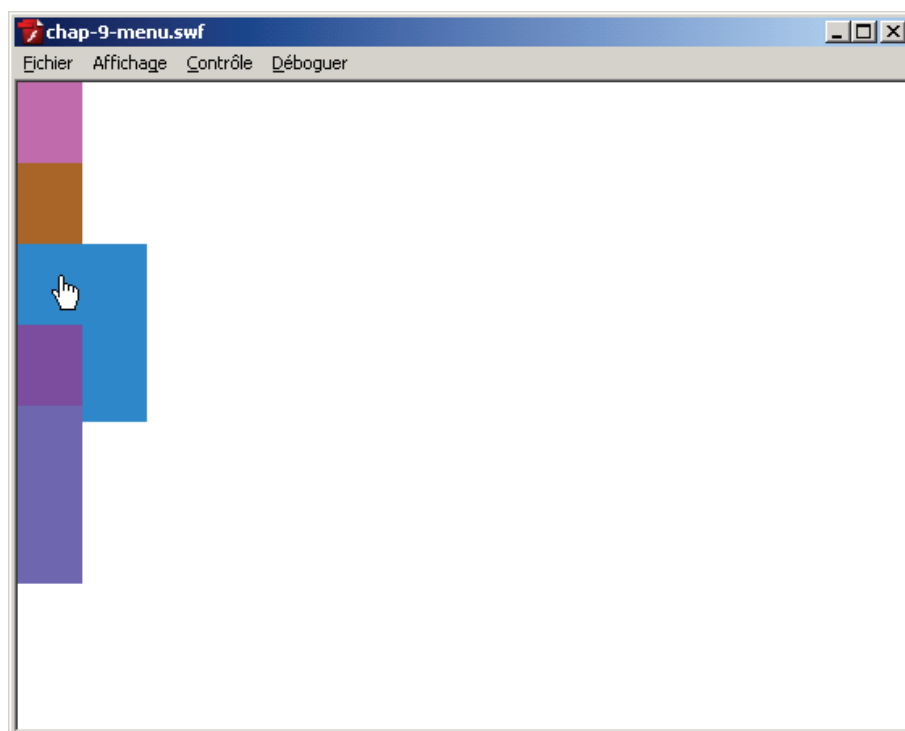
    //positionnement
    monBouton.y = 50 * i;

    // ajout au sein du conteneur
    conteneurMenu.addChild ( monBouton );

}

// affichage des boutons
addChild ( conteneurMenu );
```

Le résultat est illustré en figure 9-18 :



*Figure 9-18. Création du menu.*

La classe `Bouton` peut ainsi être réutilisée dans n'importe quel projet nécessitant un seul bouton, ou un menu.

Il manque pour le moment une fonctionnalité permettant de refermer le bouton cliqué lorsqu'un autre est sélectionné. Pour cela nous devons obligatoirement posséder une référence envers tous les boutons du menu, et pouvoir décider quels boutons refermer.

Vous souvenez-vous de la classe `Joueur` créée au cours du chapitre 8 ?

Nous avons défini un tableau stockant les références de chaque joueur créé, cela nous permettait à tout moment de savoir combien de joueurs étaient créés et de pouvoir y accéder. Nous allons reproduire le même mécanisme à l'aide d'un tableau statique.

Le tableau `tableauBoutons` stocke chaque référence de boutons :

```
package org.bytearray.ui
{
    import flash.display.Shape;
    import flash.display.Sprite;
    // import des classes Tween liées au mouvement
    import fl.transitions.Tween;
    import fl.transitions.easing.Bounce;
    // import de la classe MouseEvent
```

```
import flash.events.MouseEvent;

public class Bouton extends Sprite

{

    // stocke le fond du bouton
    private var fondBouton:Shape;
    // stocke l'objet Tween pour les différents états du bouton
    private var interpolation:Tween;
    // stocke les références aux boutons
    private static var tableauBoutons:Array = new Array();

    public function Bouton ()

    {

        // ajoute chaque instance au tableau
        Bouton.tableauBoutons.push ( this );

        // création du fond du bouton
        fondBouton = new Shape();

        // ajout à la liste d'affichage
        addChild ( fondBouton );

        // dessine le bouton
        fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
        fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

        // activation du mode bouton
        buttonMode = true;

        // création de l'objet Tween
        interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, 1, true );

        // écoute de l'événement MouseEvent.CLICK
        addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

    }

    // déclenché lors du survol du bouton
    private function survolSouris ( pEvt:MouseEvent ):void

    {

        // démarrage de l'animation
        interpolation.continueTo ( 2, 2 );

    }

}

}
```

A tout moment un bouton peut parcourir le tableau statique et accéder à ses frères et décider de les refermer.

Nous modifions la méthode `survolSouris` afin d'appeler sur chaque bouton la méthode `fermer` :

```
// déclenché lors du survol du bouton
private function survolSouris ( pEvt:MouseEvent ):void
{
    // stocke la longueur du tableau
    var lng:int = Bouton.tableauBoutons.length;

    for (var i:int = 0; i<lng; i++ ) Bouton.tableauBoutons[i].fermer();

    // démarrage de l'animation
    interpolation.continueTo ( 2, 1 );
}
```

La méthode `fermer` est privée car celle ne sera appelée qu’au sein de la classe `Bouton` :

```
// méthode permettant de refermer le bouton
private function fermer () :void
{
    // referme le bouton
    interpolation.continueTo ( 1, 1 );
}
```

Si nous testons l’animation nous remarquons que lors du survol les autres boutons ouverts se referment.

Bien entendu dans la plupart des projets ActionScript nous n’allons pas seulement créer des menus constitués de couleurs aléatoires. Il serait intéressant de pouvoir choisir la couleur de chaque bouton, pour cela nous ajoutons un paramètre afin de recevoir la couleur du bouton au sein du constructeur :

```
package org.bytearray.ui
{
    import flash.display.Shape;
    import flash.display.Sprite;
    // import des classes Tween liées au mouvement
    import fl.transitions.Tween;
    import fl.transitions.easing.Bounce;
    // import de la classe MouseEvent
    import flash.events.MouseEvent;

    public class Bouton extends Sprite
    {
        // stocke le fond du bouton
        private var fondBouton:Shape;
        // stocke l'objet Tween pour les différents états du bouton
        private var interpolation:Tween;
        // stocke les références aux boutons
        private static var tableauBoutons:Array = new Array();
        // stocke la couleur en cours du bouton
        private var couleur:Number;
```



```
public function Bouton ( pCouleur:Number )
{
    // ajoute chaque instance au tableau
    Bouton.tableauBoutons.push ( this );

    // création du fond du bouton
    fondBouton = new Shape();

    // ajout à la liste d'affichage
    addChild ( fondBouton );

    // stocke la couleur passée en paramètre
    couleur = pCouleur;

    // dessine le bouton
    fondBouton.graphics.beginFill ( couleur, 1 );
    fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

    // activation du mode bouton
    buttonMode = true;

    // création de l'objet Tween
    interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, 1, true );

    // écoute de l'événement MouseEvent.CLICK
    addEventListener ( MouseEvent.ROLL_OVER, survolSouris );
}

// déclenché lors du survol du bouton
private function survolSouris ( pEvt:MouseEvent ):void
{
    // stocke la longueur du tableau
    var lng:int = Bouton.tableauBoutons.length;

    for (var i:int = 0; i<lng; i++ )
    Bouton.tableauBoutons[i].fermer();

    // démarrage de l'animation
    interpolation.continueTo ( 2, 2 );
}

// méthode permettant de refermer le bouton
private function fermer ():void
{
    // referme le bouton
    interpolation.continueTo ( 1, 1 );
}
}
```

Désormais la classe `Bouton` accepte un paramètre pour la couleur, le code suivant crée un menu constitué de boutons rouges seulement :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

var lng:int = 5;

var monBouton:Bouton;

for (var i:int = 0; i< lng; i++ )
{
    // instantiation
    // création de boutons rouge
    monBouton = new Bouton( 0x990000 );

    //positionnement
    monBouton.y = 50 * i;

    // ajout au sein du conteneur
    conteneurMenu.addChild ( monBouton );
}

// affichage des boutons
addChild ( conteneurMenu );
```

Cela n'a pas grand intérêt, nous allons donc modifier le code actuel afin d'associer à chaque bouton, une couleur précise. Pour cela nous stockons les couleurs au sein d'un tableau qui sera parcouru, le nombre de boutons du menu sera lié à la longueur du tableau :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

// tableau de couleurs
var couleurs:Array = [0x999900, 0x881122, 0x995471, 0x332100, 0x977821];

// nombre de couleurs
var lng:int = couleurs.length;

var monBouton:Bouton;

for (var i:int = 0; i< lng; i++ )
{
    // instantiation
    monBouton = new Bouton( couleurs[i] );
```

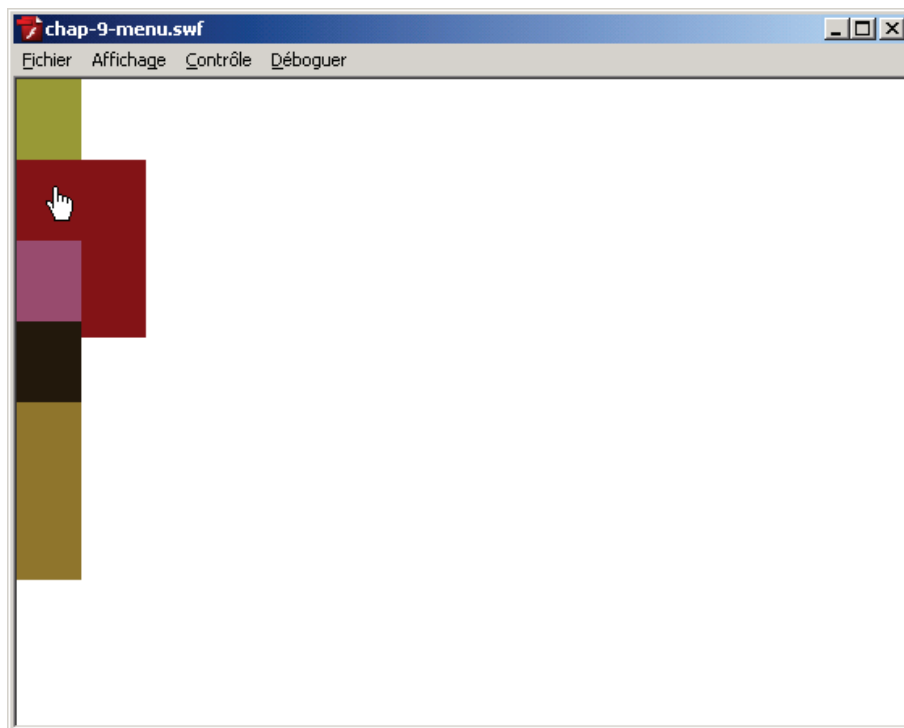
```
//positionnement
monBouton.y = 50 * i;

// ajout au sein du conteneur
conteneurMenu.addChild ( monBouton );

}

// affichage des boutons
addChild ( conteneurMenu );
```

Nous obtenons le menu illustré en figure 9-19 :



*Figure 9-19. Menu constitué de couleurs prédéfinies.*

Chaque bouton accepte une couleur lors de sa création, nous pourrions imaginer une application dans laquelle l'utilisateur choisit la couleur des boutons de son menu au sein d'une administration. Les couleurs sont alors récupérées d'une base de données puis utilisées pour créer ce menu. Nous verrons au cours du chapitre 19 intitulé *Flash Remoting* comment mettre en place cette application.

Notre menu n'est pas encore complet il serait judicieux de pouvoir spécifier la vitesse d'ouverture de chaque bouton. Souvenez-vous nous avons déjà intégré ce contrôle pour le mouvement du stylo de l'application précédente. Nos objets sont tous de la même famille mais possèdent des caractéristiques différentes, comme la couleur ou bien la vitesse d'ouverture.

Pour cela nous allons définir une propriété privée `vitesse` qui se chargera de stocker la vitesse passée à chaque bouton :

```
// stocke la vitesse d'ouverture de chaque bouton
private var vitesse:Number;
```

Puis nous définissons une méthode `affecteVitesse` qui se charge d'affecter la vitesse de manière contrôlée :

```
// gère l'affectation de la vitesse
public function affecteVitesse ( pVitesse:Number ):void
{
    // affecte la vitesse
    if ( pVitesse >= 1 && pVitesse <= 10 ) vitesse = pVitesse;

    else
    {
        trace("Erreur : Vitesse non correcte, la valeur doit être
comprise entre 1 et 10");
        vitesse = 1;
    }
}
```

Nous modifions le constructeur en ajoutant un paramètre appelé `pVitesse` afin d'appeler la méthode `affecteVitesse` avant la création de l'objet `Tween` pour initialiser la propriété `vitesse` :

```
package org.bytearray.ui
{
    import flash.display.Shape;
    import flash.display.Sprite;
    // import des classes Tween liées au mouvement
    import fl.transitions.Tween;
    import fl.transitions.easing.Bounce;
    // import de la classe MouseEvent
    import flash.events.MouseEvent;

    public class Bouton extends Sprite
    {
        // stocke le fond du bouton
        private var fondBouton:Shape;
        // stocke l'objet Tween pour les différents états du bouton
        private var interpolation:Tween;
        // stocke les références aux boutons
        private static var tableauBoutons:Array = new Array();
        // stocke la couleur en cours du bouton
        private var couleur:Number;
        // stocke la vitesse d'ouverture de chaque bouton
        private var vitesse:Number;

        public function Bouton ( pCouleur:Number, pVitesse:Number=1 )
```

```

{

    // ajoute chaque instance au tableau
    Bouton.tableauBoutons.push ( this );

    // création du fond du bouton
    fondBouton = new Shape();

    // ajout à la liste d'affichage
    addChild ( fondBouton );

    // stocke la couleur passée en paramètre
    couleur = pCouleur;

    // dessine le bouton
    fondBouton.graphics.beginFill ( couleur, 1 );
    fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

    // activation du mode bouton
    boutonMode = true;

    // affectation de la vitesse contrôlée
    affecteVitesse ( pVitesse );

    // création de l'objet Tween
    interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

    // écoute de l'événement MouseEvent.CLICK
    addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

}

// déclenché lors du survol du bouton
private function survolSouris ( pEvt:MouseEvent ):void

{

    // stocke la longueur du tableau
    var lng:int = Bouton.tableauBoutons.length;

    for (var i:int = 0; i<lng; i++ ) Bouton.tableauBoutons[i].fermer();

    // démarrage de l'animation
    interpolation.continueTo ( 2, vitesse );

}

// méthode permettant de refermer le bouton
private function fermer () :void

{

    // referme le bouton
    interpolation.continueTo ( 1, vitesse );

}

// gère l'affectation de la vitesse
public function affecteVitesse ( pVitesse:Number ):void

```

```
{  
  
    // affecte la vitesse  
    if ( pVitesse >= 1 && pVitesse <= 10 ) vitesse = pVitesse;  
  
    else  
  
        {  
            trace("Erreur : Vitesse non correcte, la valeur doit être  
comprise entre 1 et 10");  
            vitesse = 1;  
        }  
  
}  
  
}
```

Dans le code suivant, nous créons un menu composé de boutons dont nous pouvons choisir la vitesse d'ouverture :

```
// import de la classe Bouton  
import org.bytearray.ui.Bouton;  
  
// création d'un conteneur pour le menu  
var conteneurMenu:Sprite = new Sprite();  
  
// tableau de couleurs  
var couleurs:Array = [0x999900, 0x881122, 0x995471, 0x332100, 0x977821];  
  
// nombre de couleurs  
var lng:int = couleurs.length;  
  
var monBouton:Bouton;  
  
for (var i:int = 0; i< lng; i++ )  
{  
  
    // instantiation  
    monBouton = new Bouton( couleurs[i], 1 );  
  
    //positionnement  
    monBouton.y = 50 * i;  
  
    // ajout au sein du conteneur  
    conteneurMenu.addChild ( monBouton );  
}  
  
// affichage du menu  
addChild ( conteneurMenu );
```

Au cas où la vitesse passée ne serait pas correcte, le menu continue de fonctionner et un message d'erreur est affiché :

```
// affiche : Erreur : Vitesse non correcte, la valeur  
// doit être comprise entre 1 et 10  
var monBouton:Bouton = new Bouton( couleurs[i], 0 );
```

Généralement, les boutons d'un menu contiennent une légende, nous allons donc ajouter un champ texte à chaque bouton. Pour cela nous importons les classes `flash.text.TextField` et `flash.text.TextFieldAutoSize` :

```
// import de la classe TextField et TextFieldAutoSize
import flash.text.TextField;
import flash.text.TextFieldAutoSize;
```

Nous créons une propriété `legende` afin de référencer la légende :

```
// légende du bouton
private var legende:TextField;
```

Puis nous ajoutons le champ au bouton au sein du constructeur :

```
public function Bouton ( pCouleur:Number, pVitesse:Number=1 )
{

    // ajoute chaque instance au tableau
    Bouton.tableauBoutons.push ( this );

    // création du fond du bouton
    fondBouton = new Shape();

    // ajout à la liste d'affichage
    addChild ( fondBouton );

    // crée le champ texte
    legende = new TextField();

    // redimensionnement automatique du champ texte
    legende.autoSize = TextFieldAutoSize.LEFT;

    // rend le champ texte non sélectionnable
    legende.selectable = false;

    // ajout à la liste d'affichage
    addChild ( legende );

    // stocke la couleur passée en paramètre
    couleur = pCouleur;

    // dessine le bouton
    fondBouton.graphics.beginFill ( couleur, 1 );
    fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

    // activation du mode bouton
    buttonMode = true;

    // affectation de la vitesse contrôlée
    affecteVitesse ( pVitesse );

    // création de l'objet Tween
    interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

    // écoute de l'événement MouseEvent.CLICK
    addEventListener ( MouseEvent.ROLL_OVER, survolSouris );
```

```
| }
```

Si nous testons notre menu, nous remarquons que le champ texte imbriqué dans le bouton reçoit les entrées souris et entre en conflit avec l'enveloppe principale du bouton.

Afin de désactiver les objets enfants du bouton nous passons la valeur `false` à la propriété `mouseChildren` du bouton :

```
| // désactivation des objets enfants  
| mouseChildren = false;
```

Souvenez-vous, lors du chapitre 7, nous avons vu que la propriété `mouseChildren` permettait de désactiver les événements souris auprès des objets enfants.

Nous ajoutons un paramètre au constructeur de la classe `Bouton` afin d'accueillir le texte affiché par la légende :

```
public function Bouton ( pCouleur:Number, pVitesse:Number=1,  
pLegende:String="Légende" )  
{  
  
    // ajoute chaque instance au tableau  
    Bouton.tableauBoutons.push ( this );  
  
    // création du fond du bouton  
    fondBouton = new Shape();  
  
    // ajout à la liste d'affichage  
    addChild ( fondBouton );  
  
    // crée le champ texte  
    legende = new TextField();  
  
    // redimensionnement automatique du champ texte  
    legende.autoSize = TextFieldAutoSize.LEFT;  
  
    // rend le champ texte non sélectionnable  
    legende.selectable = false;  
  
    // ajout à la liste d'affichage  
    addChild ( legende );  
  
    // affecte la légende  
    legende.text = pLegende;  
  
    // stocke la couleur passée en paramètre  
    couleur = pCouleur;  
  
    // dessine le bouton  
    fondBouton.graphics.beginFill ( couleur, 1 );  
    fondBouton.graphics.drawRect ( 0, 0, 40, 110 );  
  
    // activation du mode bouton  
    boutonMode = true;  
  
    // désactivation des objets enfants
```



```
mouseChildren = false;

// affectation de la vitesse contrôlée
affecteVitesse ( pVitesse );

// création de l'objet Tween
interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

// écoute de l'événement MouseEvent.CLICK
addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

}
```

Les données utilisées afin de générer un menu proviennent généralement d'un flux XML ou de *Flash Remoting* et sont très souvent formatées sous forme de tableau associatif.

Nous allons réorganiser nos données afin d'utiliser un tableau associatif plutôt que plusieurs tableaux séparés :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

// tableau associatif contenant les données
var donnees:Array = new Array();

donnees.push ( { legende : "Accueil", vitesse : 1, couleur : 0x999900 } );
donnees.push ( { legende : "Photos", vitesse : 1, couleur : 0x881122 } );
donnees.push ( { legende : "Blog", vitesse : 1, couleur : 0x995471 } );
donnees.push ( { legende : "Liens", vitesse : 1, couleur : 0x332100 } );
donnees.push ( { legende : "Forum", vitesse : 1, couleur : 0x977821 } );

// nombre de rubriques
var lng:int = donnees.length;

var monBouton:Bouton;
var legende:String;
var couleur:Number;
var vitesse:Number;

for (var i:int = 0; i < lng; i++ )
{

    // récupération des infos
    legende = donnees[i].legende;
    couleur = donnees[i].couleur;
    vitesse = donnees[i].vitesse;

    // création des boutons
    monBouton = new Bouton( couleur, vitesse, legende );

    // positionnement
    monBouton.y = 50 * i;

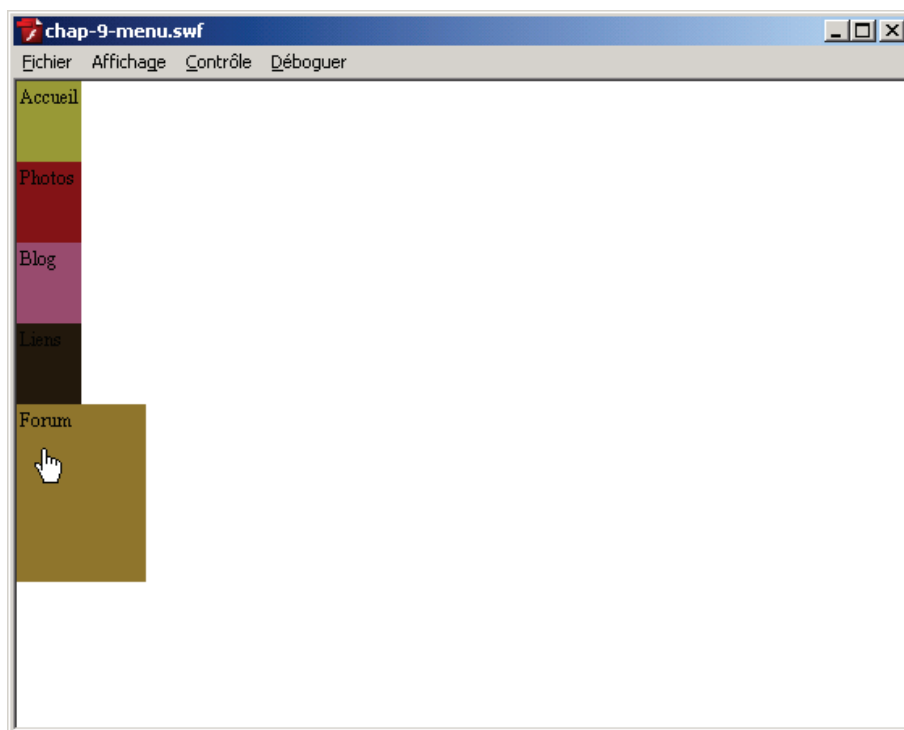
    // ajout au sein du conteneur
    conteneurMenu.addChild ( monBouton );
}
```

```

    }
    // affichage du menu
    addChild ( conteneurMenu );

```

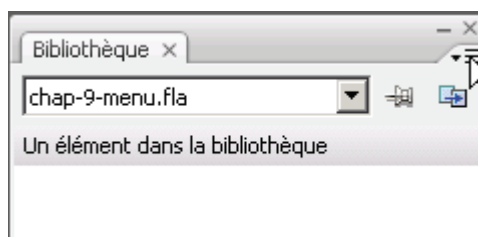
En testant notre menu, nous obtenons le résultat illustré en figure 9-20 :



*Figure 9-20. Boutons avec légendes.*

Le texte de légende n'est pas correctement formaté, pour y remédier nous utilisons la classe `flash.text.TextFormat` ainsi que la classe `flash.text.Font`. Nous reviendrons plus en profondeur sur le formatage du texte au cours du chapitre 16 intitulé *Le texte*.

Afin d'intégrer une police au sein de la bibliothèque nous cliquons sur l'icône prévue illustrée en figure 9-21 :



*Figure 9-21. Options de la bibliothèque.*

Le panneau d'options de bibliothèque s'ouvre, nous sélectionnons *Nouvelle Police* comme l'illustre la figure 9-22 :

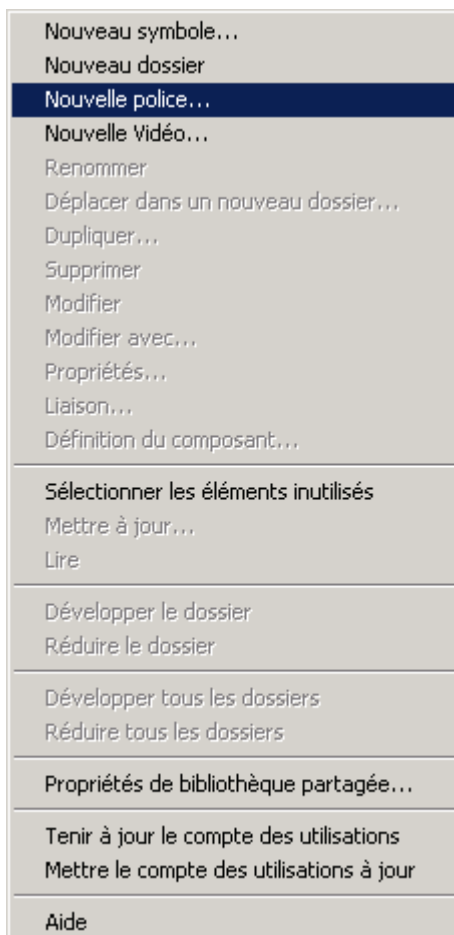


Figure 9-22. Insertion de police dans la bibliothèque.

Une fois sélectionnée, le panneau *Propriétés des symboles de police* s'affiche, ce dernier permet de sélectionner la police qui sera embarquée dans l'animation. Nous sélectionnons pour notre exemple la police *Trebuchet MS*, puis nous donnons un nom de classe à la police, nous conservons *Police 1*.

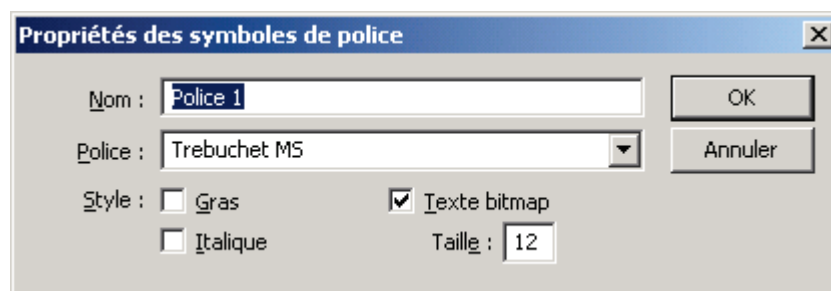
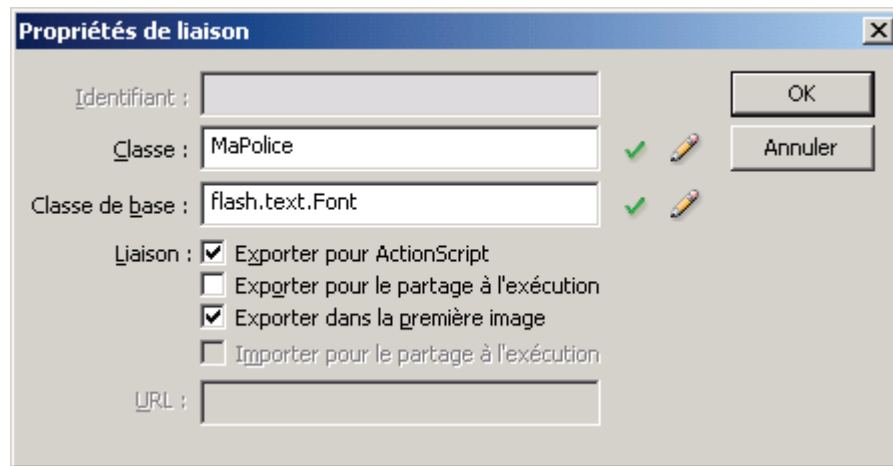


Figure 9-23. Propriétés des symboles de police.

Une fois validé, la police apparaît dans la bibliothèque, il ne nous reste plus qu'à l'utiliser au sein de nos champs texte contenus dans chaque bouton. Pour cela il faut lier cette police comme pour un symbole classique. En faisant un clic droit sur celle-ci, nous sélectionnons l'option *Liaisons* le panneau *Propriétés de liaison* s'ouvre comme l'illustre la figure 9-24 :



*Figure 9-24. Panneau propriétés de liaison.*

Nous choisissons `MaPolice` comme nom de classe, celle-ci va automatiquement être créée par Flash et héritera de la classe `flash.text.Font`.

Nous définissons une nouvelle propriété `formatage` afin de stocker l'objet `TextFormat` :

```
// formatage des légendes
private var formatage:TextFormat;
```

Nous importons la classe `flash.text.TextFormat` :

```
import flash.text.TextFormat;
```

Puis nous modifions le constructeur de la classe `Bouton` afin d'affecter le formatage et indiquer au champ texte d'utiliser la police embarquée :

```
public function Bouton ( pCouleur:Number, pVitesse:Number, pLegende:String
)
{
    // ajoute chaque instance au tableau
    Bouton.tableauBoutons.push ( this );

    // création du fond du bouton
    fondBouton = new Shape();

    // ajout à la liste d'affichage
    addChild ( fondBouton );
```

```
// crée le champ texte
legende = new TextField();

// redimensionnement automatique du champ texte
legende.autoSize = TextFieldAutoSize.LEFT;

// ajout à la liste d'affichage
addChild ( legende );

// affecte la légende
legende.text = pLegende;

// active l'utilisation de police embarquée
legende.embedFonts = true;

// crée un objet de formatage
formatage = new TextFormat();

// taille de la police
formatage.size = 12;

// instanciation de la police embarquée
var police:MaPolice = new MaPolice();

// affectation de la police au formatage
formatage.font = police.fontName;

// affectation du formatage au champ texte
legende.setTextFormat ( formatage );

// rend le champ texte non sélectionnable
legende.selectable = false;

// stocke la couleur passée en paramètre
couleur = pCouleur;

// dessine le bouton
fondBouton.graphics.beginFill ( couleur, 1 );
fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

// activation du mode bouton
buttonMode = true;

// désactivation des objets enfants
mouseChildren = false;

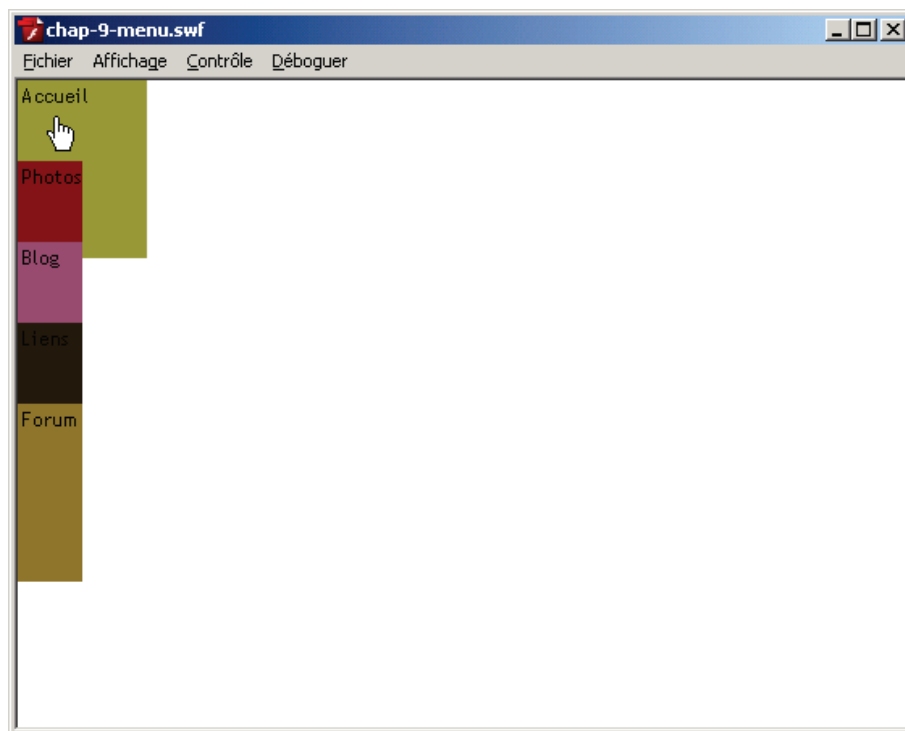
// affectation de la vitesse contrôlée
affecteVitesse ( pVitesse );

// création de l'objet Tween
interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

// écoute de l'événement MouseEvent.CLICK
addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

}
```

En testant le code précédent, chaque bouton possède désormais une légende intégrant les contours de police :



*Figure 9-25. Champs texte formatés.*

Il ne nous reste plus qu'à intégrer une gestion de la largeur et hauteur de chaque bouton. Pour cela nous ajoutons au constructeur de la classe **Bouton** deux paramètres **pLargeur** et **pHauteur** :

```

    public function Bouton ( pLargeur:Number, pHauteur:Number, pCouleur:Number,
    pVitesse:Number, pLegende:String )
    {

        // ajoute chaque instance au tableau
        Bouton.tableauBoutons.push ( this );

        // création du fond du bouton
        fondBouton = new Shape();

        // ajout à la liste d'affichage
        addChild ( fondBouton );

        // crée le champ texte
        legende = new TextField();

        // redimensionnement automatique du champ texte
        legende.autoSize = TextFieldAutoSize.LEFT;

        // ajout à la liste d'affichage
        addChild ( legende );

        // affecte la légende
        legende.text = pLegende;

        // active l'utilisation de police embarquée
        legende.embedFonts = true;
    }

```

```

// crée un objet de formatage
formatage = new TextFormat();

// taille de la police
formatage.size = 12;

// instanciation de la police embarquée
var police:MaPolice = new MaPolice();

// affectation de la police au formatage
formatage.font = police.fontName;

// affectation du formatage au champ texte
legende.setTextFormat ( formatage );

// rend le champ texte non sélectionnable
legende.selectable = false;

// stocke la couleur passée en paramètre
couleur = pCouleur;

// dessine le bouton
fondBouton.graphics.beginFill ( couleur, 1 );
fondBouton.graphics.drawRect ( 0, 0, pLargeur, pHauteur );

// activation du mode bouton
buttonMode = true;

// désactivation des objets enfants
mouseChildren = false;

// affectation de la vitesse contrôlée
affecteVitesse ( pVitesse );

// création de l'objet Tween
interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

// écoute de l'événement MouseEvent.CLICK
addEventListener ( MouseEvent.ROLL_OVER, survolSouris );
}

```

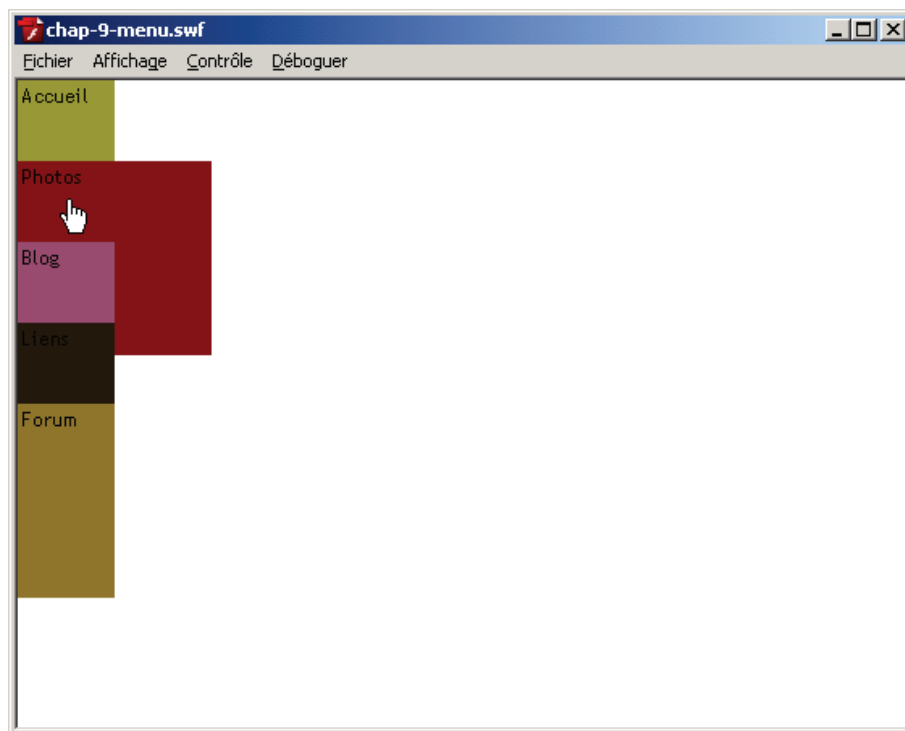
Puis nous passons les valeurs voulues lors de la création de chaque bouton :

```

// création des boutons
monBouton = new Bouton( 60, 120, couleur, vitesse, legende );

```

La figure 9-26 illustre le résultat final :



*Figure 9-26. Boutons aux dimensions dynamiques.*

Nous obtenons un menu simple à mettre en place et facilement modifiable. Nous pourrions rajouter d'autres fonctionnalités comme une adaptation automatique de la largeur du bouton par rapport à la légende. En réalité, nous pourrions ne jamais nous arrêter !

Dans beaucoup de projets, les boutons d'un menu sont généralement liés à des SWF. Lorsque le bouton est cliqué, le SWF correspondant est chargé, cela permet un meilleur découpage du site et un chargement à la demande. Nous allons modifier la classe `Bouton` afin de pouvoir stocker dans chaque bouton le nom du SWF correspondant. Nous ajoutons une propriété `swf` :

```
// swf associé
private var swf:String;
```

Puis nous modifions le constructeur afin d'accueillir le nom du SWF associé :

```
public function Bouton ( pLargeur:Number, pHauteur:Number, pSWF:String,
pCouleur:Number, pVitesse:Number, pLegende:String )
{
    // ajoute chaque instance au tableau
    Bouton.tableauBoutons.push ( this );

    // création du fond du bouton
    fondBouton = new Shape();
```



```
// ajout à la liste d'affichage
addChild ( fondBouton );

// crée le champ texte
legende = new TextField();

// redimensionnement automatique du champ texte
legende.autoSize = TextFieldAutoSize.LEFT;

// ajout à la liste d'affichage
addChild ( legende );

// affecte la légende
legende.text = pLegende;

// active l'utilisation de police embarquée
legende.embedFonts = true;

// crée un objet de formatage
formatage = new TextFormat();

// taille de la police
formatage.size = 12;

// instanciation de la police embarquée
var police:MaPolice = new MaPolice();

// affectation de la police au formatage
formatage.font = police.fontName;

// affectation du formatage au champ texte
legende.setTextFormat ( formatage );

// rend le champ texte non sélectionnable
legende.selectable = false;

// stocke la couleur passée en paramètre
couleur = pCouleur;

// stocke le nom du SWF associé
swf = pSWF;

// dessine le bouton
fondBouton.graphics.beginFill ( couleur, 1 );
fondBouton.graphics.drawRect ( 0, 0, pLargeur, pHauteur );

// activation du mode bouton
buttonMode = true;

// désactivation des objets enfants
mouseChildren = false;

// affectation de la vitesse contrôlée
affecteVitesse ( pVitesse );

// création de l'objet Tween
interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

// écoute de l'événement MouseEvent.CLICK
addEventListener ( MouseEvent.ROLL_OVER, survolSouris );
```

```
| }  
|
```

Puis nousinstancions les boutons du menu :

```
// import de la classe Bouton  
import org.bytearray.ui.Bouton;  
  
// création d'un conteneur pour le menu  
var conteneurMenu:Sprite = new Sprite();  
  
// tableau associatif contenant les données  
var donnees:Array = new Array();  
  
donnees.push ( { legende : "Accueil", vitesse : 1, swf : "accueil.swf",  
couleur : 0x999900 } );  
donnees.push ( { legende : "Photos", vitesse : 1, swf : "photos.swf", couleur :  
: 0x881122 } );  
donnees.push ( { legende : "Blog", vitesse : 1, swf : "blog.swf", couleur :  
0x995471 } );  
donnees.push ( { legende : "Liens", vitesse : 1, swf : "liens.swf", couleur :  
0xCC21FF } );  
donnees.push ( { legende : "Forum", vitesse : 1, swf : "forum.swf", couleur :  
0x977821 } );  
  
// nombre de rubriques  
var lng:int = donnees.length;  
  
var monBouton:Bouton;  
var legende:String;  
var couleur:Number;  
var vitesse:Number;  
var swf:String;  
  
for (var i:int = 0; i< lng; i++ )  
{  
  
    // récupération des infos  
    legende = donnees[i].legende;  
    couleur = donnees[i].couleur;  
    vitesse = donnees[i].vitesse;  
    swf = donnees[i].swf;  
  
    // création des boutons  
    monBouton = new Bouton( 60, 120, swf, couleur, vitesse, legende );  
  
    // positionnement  
    monBouton.y = 50 * i;  
  
    // ajout au sein du conteneur  
    conteneurMenu.addChild ( monBouton );  
  
}  
  
// affichage du menu  
addChild ( conteneurMenu );
```

Chaque bouton est ainsi lié à un SWF. Lors du chapitre 7 nous avons créé un menu dynamique qui nous redirigeait à une adresse spécifique en ouvrant une nouvelle fenêtre navigateur. Pour cela nous stockions

le lien au sein d'une propriété du bouton, puis au moment du clic nous accédions à celle-ci.

Voici le code final de la classe `Bouton` :

```
package org.bytearray.ui

{

    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.text.Font;
    import flash.text.TextFormat;
    // import des classes liées Tween au mouvement
    import fl.transitions.Tween;
    import fl.transitions.easing.Bounce;
    // import de la classe MouseEvent
    import flash.events.MouseEvent;
    // import de la classe TextField et TextFieldAutoSize
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;

    public class Bouton extends Sprite

    {

        // stocke le fond du bouton
        private var fondBouton:Shape;
        // stocke l'objet Tween pour les différents états du bouton
        private var interpolation:Tween;
        // stocke les références aux boutons
        private static var tableauBoutons:Array = new Array();
        // stocke la couleur en cours du bouton
        private var couleur:Number;
        // stocke la vitesse d'ouverture de chaque bouton
        private var vitesse:Number;
        // légende du bouton
        private var legende:TextField;
        // formatage des légendes
        private var formatage:TextFormat;
        // swf associé
        private var swf:String;

        public function Bouton ( pLargeur:Number, pHauteur:Number,
            pSWF:String, pCouleur:Number, pVitesse:Number, pLegende:String )

        {

            // ajoute chaque instance au tableau
            Bouton.tableauBoutons.push ( this );

            // création du fond du bouton
            fondBouton = new Shape();

            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // crée le champ texte
            legende = new TextField();

            // redimensionnement automatique du champ texte
```

```

        legende.autoSize = TextFieldAutoSize.LEFT;

        // ajout à la liste d'affichage
        addChild ( legende );

        // affecte la légende
        legende.text = pLegende;

        // active l'utilisation de police embarquée
        legende.embedFonts = true;

        // crée un objet de formatage
        formatage = new TextFormat();

        // taille de la police
        formatage.size = 12;

        // instantiation de la police embarquée
        var police:MaPolice = new MaPolice();

        // affectation de la police au formatage
        formatage.font = police.fontName;

        // affectation du formatage au champ texte
        legende.setTextFormat ( formatage );

        // rend le champ texte non sélectionnable
        legende.selectable = false;

        // stocke la couleur passée en paramètre
        couleur = pCouleur;

        // stocke le nom du SWF
        swf = pSWF;

        // dessine le bouton
        fondBouton.graphics.beginFill ( couleur, 1 );
        fondBouton.graphics.drawRect ( 0, 0, pLargeur, pHauteur );

        // activation du mode bouton
        buttonMode = true;

        // désactivation des objets enfants
        mouseChildren = false;

        // affectation de la vitesse contrôlée
        affecteVitesse ( pVitesse );

        // création de l'objet Tween
        interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, vitesse, true );

        // écoute de l'événement MouseEvent.CLICK
        addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

    }

    // déclenché lors du survol du bouton
    private function survolSouris ( pEvt:MouseEvent ):void

    {
        // stocke la longueur du tableau

```

```
        var lng:int = Bouton.tableauBoutons.length;

        for (var i:int = 0; i<lng; i++ )
        Bouton.tableauBoutons[i].fermer();

        // démarrage de l'animation
        interpolation.continueTo ( 2, vitesse );

    }

    // méthode permettant de refermer le bouton
    private function fermer ():void

    {
        // referme le bouton
        interpolation.continueTo ( 1, vitesse );

    }

    // gère l'affectation de la vitesse
    public function affecteVitesse ( pVitesse:Number ):void

    {

        // affecte la vitesse
        if ( pVitesse >= 1 && pVitesse <= 10 ) vitesse = pVitesse;

        else

        {
            trace("Erreur : Vitesse non correcte, la valeur doit être
comprise entre 1 et 10");
            vitesse = 1;

        }

    }

}

}
```

Cette approche fonctionne sans problème et convient dans beaucoup de situations, en revanche lors de l'utilisation de classes personnalisées comme dans cette application nous allons externaliser les informations nécessaires grâce au modèle événementiel. De cette manière les objets « parleront » entre eux de manière faiblement couplée.

Nous allons épouser le même modèle que les objets natifs d'ActionScript 3, chaque bouton pourra diffuser un événement spécifique nous renseignant sur sa couleur, sa vitesse ainsi que le SWF correspondant.

Certains d'entre vous ont peut être déjà deviné vers quelle nouvelle notion nous nous dirigeons, en route pour le chapitre suivant intitulé *Diffusion d'événements personnalisés*.

# 10

## Diffusion d'événements personnalisés

<b>L'HISTOIRE</b> .....	1
<b>LA CLASSE EVENTDISPATCHER</b> .....	2
MISE EN APPLICATION.....	4
CHOISIR UN NOM D'ÉVÉNEMENT.....	9
<b>ETENDRE EVENTDISPATCHER</b> .....	11
<b>STOCKER EVENTDISPATCHER</b> .....	14
PASSER DES INFORMATIONS.....	19
MENU ET ÉVÉNEMENT PERSONNALISÉ .....	23

### L'histoire

Dans une application ActionScript, la communication inter objets se réalise majoritairement par la diffusion d'événements natifs ou personnalisés. Mais qu'est ce qu'un événement personnalisé ?

Il s'agit d'un événement qui n'existe pas au sein du lecteur Flash mais que nous ajoutons afin de gérer les différentes interactions entre nos objets. Depuis très longtemps ActionScript intègre différentes classes permettant de diffuser nos propres événements.

La première fut `AsBroadcaster` intégrée depuis le lecteur Flash 5, son implémentation native l'avait rendu favorite des développeurs ActionScript. Sa remplaçante `BroadcasterMX` introduite plus tard par Flash MX et codée en ActionScript fut moins bien accueillie. Aujourd'hui ces classes n'existent plus en ActionScript 3 et sont remplacées par la classe native `flash.events.EventDispatcher`.

Au sein d'une interface graphique nous pourrions imaginer un bouton diffusant en événement indiquant son état actif ou inactif. Une classe générant des fichiers PDF pourrait diffuser des événements relatifs au processus de création du fichier. Ainsi, la diffusion d'événements personnalisés constitue la suite logique à la création d'objets personnalisés.

## La classe `EventDispatcher`

Comme nous le voyons depuis le début de l'ouvrage, ActionScript 3 est un langage basé sur un modèle événementiel appelé *Document Object Model*. Celui-ci repose sur la classe `EventDispatcher` dont toutes les classes natives de l'API du lecteur Flash héritent.

Dans le code suivant nous voyons la relation entre la classe `EventDispatcher` et deux classes graphiques :

```
var monSprite:Sprite = new Sprite();

// affiche : true
trace( monSprite is EventDispatcher );

var monClip:MovieClip = new MovieClip();

// affiche : true
trace( monClip is EventDispatcher );
```

Rappelez-vous, toutes les classes issues du paquetage `flash` sont des sous-classes d'`EventDispatcher` et possèdent donc ce type commun.

Nous créons un `Sprite` puis nous écoutons un événement personnalisé auprès de ce dernier :

```
// création d'un Sprite
var monSprite:Sprite = new Sprite();

// écoute de l'événement monEvent
monSprite.addEventListener ( "monEvenement", ecouteur );

// fonction écouteur
function ecouteur ( pEvt:Event ):void
{
    trace( pEvt );
}
```

La classe `Sprite` ne diffuse pas par défaut d'événement nommé `monEvenement`, mais nous pouvons le diffuser simplement grâce à la méthode `dispatchEvent` dont voici la signature :

```
public function dispatchEvent(event:Event):Boolean
```

En ActionScript 2 la méthode `dispatchEvent` diffusait un objet événementiel non typé. Nous utilisons généralement un objet littéral auquel nous ajoutons différentes propriétés manuellement.

En ActionScript 3, afin de diffuser un événement nous devons créer un objet événementiel, représenté par une instance de la classe `flash.events.Event` :

```
// création de l'objet événementiel
var objetEvenementiel:Event = new Event (type, bubbles, cancelable);
```

Le constructeur de la classe `Event` accepte trois paramètres :

- `type` : le nom de l'événement à diffuser.
- `bubbles` : indique si l'événement participe à la phase de remontée.
- `cancelable` : indique si l'événement peut être annulé.

Dans la plupart des situations nous n'utilisons que le premier paramètre `type` de la classe `Event`.

Une fois l'objet événementiel créé, nous le passons à la méthode `dispatchEvent` :

```
// création d'un sprite
var monSprite:Sprite = new Sprite();

// écoute de l'événement monEvent
monSprite.addEventListener ( "monEvenement", ecouteur );

// fonction écouteur
function ecouteur (pEvt:Event):void
{
    // affiche [Event type="monEvenement" bubbles=false cancelable=false
    eventPhase=2]
    trace( pEvt );
}

// création de l'objet événementiel
var objetEvenementiel:Event = new Event ("monEvenement", bubbles,
cancelable);

// nous diffusons l'événement monEvenement
monSprite.dispatchEvent (objetEvenementiel);
```

Généralement, nous ne créons pas l'objet événementiel séparément, nous l'instancions directement en paramètre de la méthode `dispatchEvent` :

```
// création d'un Sprite
var monSprite:Sprite = new Sprite();

// écoute de l'événement monEvent
monSprite.addEventListener ("monEvenement", ecouteur );
```



```
// fonction écouteur
function ecouteur ( pEvt:Event ):void
{
    // affiche [Event type="monEvenement" bubbles=false cancelable=false
    eventPhase=2]
    trace( pEvt );
}

// nous diffusons l'événement monEvenement
monSprite.dispatchEvent ( new Event ("monEvenement") );
```

Cet exemple nous montre la facilité avec laquelle nous pouvons diffuser un événement en ActionScript 3.

## A retenir

- Le modèle événementiel ActionScript 3 repose sur la classe `EventDispatcher`.
- Toutes les classes issues du paquetage `flash` peuvent diffuser des événements natifs ou personnalisés.
- Afin de diffuser un événement, nous utilisons la méthode `dispatchEvent`.
- La méthode `dispatchEvent` accepte comme paramètre une instance de la classe `Event`.

## Mise en application

Nous allons développer une classe permettant à un symbole de se déplacer dans différentes directions. Lorsque celui-ci arrive à destination nous souhaitons diffuser un événement approprié.

A côté d'un nouveau document Flash CS3, nous sauvons une classe nommée `Balle.as` contenant le code suivant :

```
package
{
    import flash.display.Sprite;

    // la classe Balle étend la classe Sprite
    public class Balle extends Sprite
    {
        public function Balle ()
        {
            trace( this );
        }
    }
}
```

```
    }  
  }  
}
```

Nous lions notre symbole de forme circulaire à celle-ci par le panneau *Propriétés de liaison*. Puis nousinstancions le symbole :

```
// création du symbole  
// affiche : [object Balle]  
var maBalle:Balle = new Balle();  
  
// ajout à la liste d'affichage  
addChild ( maBalle );
```

A chaque clic souris sur la scène, la balle doit se diriger vers le point cliqué. Nous devons donc écouter l'événement `MouseEvent.CLICK` de manière globale auprès de l'objet `Stage`.

Nous avons vu au cours du précédent chapitre comment accéder de manière sécurisée à l'objet `Stage`. Nous intégrons le même mécanisme dans la classe `Balle` :

```
package  
  
{  
  
    import flash.display.Sprite;  
    import flash.events.MouseEvent;  
    import flash.events.Event;  
  
    // la classe Balle étend la classe Sprite  
    public class Balle extends Sprite  
    {  
  
        public function Balle ()  
        {  
            // écoute de l'événement Event.ADDED_TO_STAGE  
            addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );  
        }  
  
        private function ajoutAffichage( pEvt:Event ):void  
        {  
            // écoute de l'événement MouseEvent.CLICK  
            stage.addEventListener ( MouseEvent.CLICK, clicSouris );  
        }  
  
        private function clicSouris ( pEvt:MouseEvent ):void  
        {
```

```

        // affiche : [MouseEvent type="click" bubbles=true
cancelable=false eventPhase=2 localX=81 localY=127 stageX=81 stageY=127
relatedObject=null ctrlKey=false altKey=false shiftKey=false delta=0]
        trace( pEvt );

    }

}

}

```

A chaque clic sur la scène, l'événement `MouseEvent.CLICK` est diffusé, la fonction écouteur `clicSouris` est alors déclenchée.

Nous allons intégrer à présent la notion de mouvement. Nous définissons deux propriétés `sourisX` et `sourisY` au sein de la classe. Celles-ci vont nous permettre de stocker la position de la souris :

```

// stocke les coordonnées de la souris
private var sourisX:Number;
private var sourisY:Number;

```

Puis nous modifions la méthode `clicSouris` afin d'affecter ces propriétés :

```

package
{

    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.events.Event;

    // la classe Balle étend la classe Sprite
    public class Balle extends Sprite

    {

        // stocke les coordonnées de la souris
        private var sourisX:Number;
        private var sourisY:Number;

        public function Balle ()

        {

            // écoute de l'événement Event.ADDED_TO_STAGE
            addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );

        }

        private function ajoutAffichage( pEvt:Event ):void

        {

            // écoute de l'événement MouseEvent.CLICK
            stage.addEventListener ( MouseEvent.CLICK, clicSouris );

        }

        private function clicSouris ( pEvt:MouseEvent ):void

```

```
        {  
            // affecte les coordonnées aux propriétés  
            sourisX = pEvt.stageX;  
            sourisY = pEvt.stageY;  
        }  
    }  
}
```

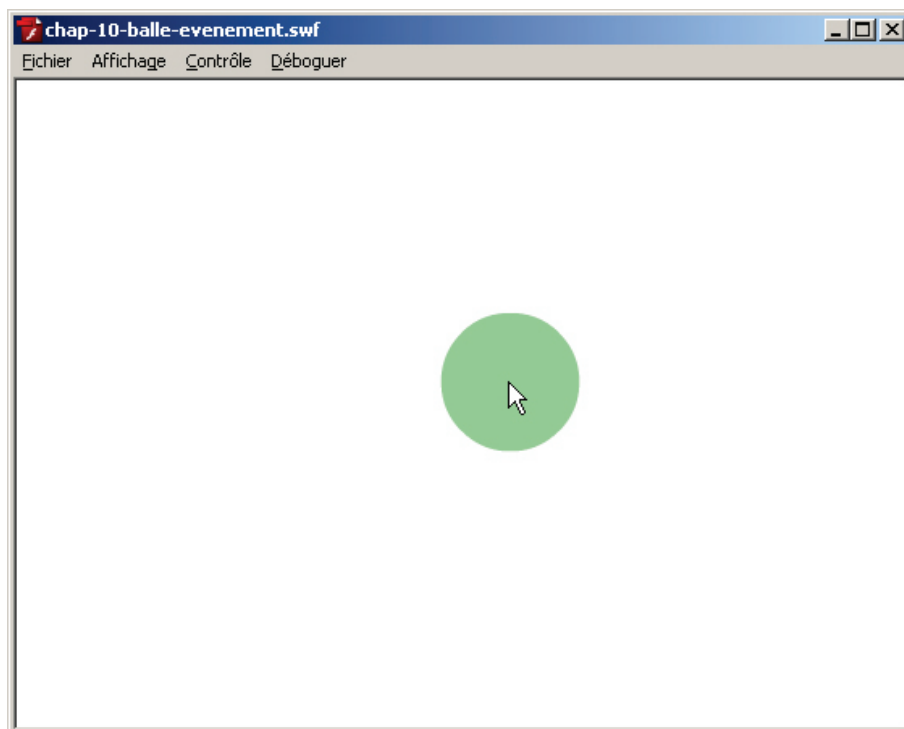
Enfin, nous déclenchons le mouvement en écoutant l'événement `Event.ENTER_FRAME` hérité de la classe `Sprite` :

```
package  
{  
    import flash.display.Sprite;  
    import flash.events.MouseEvent;  
    import flash.events.Event;  
  
    // la classe Balle étend la classe Sprite  
    public class Balle extends Sprite  
    {  
        // stocke les coordonnées de la souris  
        private var sourisX:Number;  
        private var sourisY:Number;  
  
        public function Balle ()  
        {  
            // écoute de l'événement Event.ADDED_TO_STAGE  
            addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );  
        }  
  
        private function ajoutAffichage( pEvt:Event ):void  
        {  
            // écoute de l'événement MouseEvent.CLICK  
            stage.addEventListener ( MouseEvent.CLICK, clicSouris );  
        }  
  
        private function clicSouris ( pEvt:MouseEvent ):void  
        {  
            // affecte les coordonnées aux propriétés  
            sourisX = pEvt.stageX;  
            sourisY = pEvt.stageY;  
  
            addEventListener ( Event.ENTER_FRAME, mouvement );  
        }  
    }  
}
```

```
private function mouvement ( pEvt:Event ):void
{
    // évalue la destination x et y
    var destinationX:Number = ( sourisX - width/2 );
    var destinationY:Number = ( sourisY - height/2 );

    // déplace la balle avec un effet de ralentissement (inertie)
    x -= (x - destinationX)*.1;
    y -= (y - destinationY)*.1;
}
}
```

Si nous testons notre animation, la balle se déplace à l'endroit cliqué avec un effet de ralenti. La figure 10.1 illustre le comportement.



*Figure 10.1 Déplacement position clic souris.*

Lorsqu'un mouvement entre en jeu, nous souhaitons généralement savoir quand est ce qu'il se termine. La classe `Tween` diffuse par défaut tous les événements nécessaires à la synchronisation d'une animation. Mais comment faire dans notre cas pour diffuser notre propre événement ?

La classe `Sprite` hérite de la classe `EventDispatcher` et peut donc diffuser n'importe quel événement. Au sein de la méthode

`mouvement` nous testons si la différence entre la position en cours et la destination est inférieure à 1. Si c'est le cas, cela signifie que nous sommes arrivés à destination.

```
private function mouvement ( pEvt:Event ):void
{
    // évalue la destination x et y
    var destinationX:Number = ( sourisX - width/2);
    var destinationY:Number = ( sourisY - height/2);

    // déplace la balle avec un effet de ralentissement (inertie)
    x -= (x - destinationX)*.1;
    y -= (y - destinationY)*.1;

    if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY
) < 1 )
    {
        removeEventListener ( Event.ENTER_FRAME, mouvement );

        trace ("arrivé à destination !");
    }
}
```

La méthode `Math.abs` nous permet de rendre la distance absolue, car une distance entre deux points est toujours positive.

Nous supprimons l'écoute de l'événement `Event.ENTER_FRAME` lorsque la balle arrive à destination afin d'optimiser les ressources, puis nous affichons un message indiquant que la balle est arrivée.

En testant notre animation, nous remarquons que le message indiquant l'arrivée est bien déclenché, mais pour l'instant aucun événement n'est diffusé.

## Choisir un nom d'événement

Lorsqu'un objet diffuse un événement nous devons nous assurer que son nom soit *simple* et *intuitif* pour les personnes utilisant la classe. Dans notre exemple nous allons diffuser un événement `mouvementTermine`.

Nous modifions la méthode `mouvement` afin que celle-ci le diffuse :

```
private function mouvement ( pEvt:Event ):void
{
    // évalue la destination x et y
    var destinationX:Number = ( sourisX - width/2);
    var destinationY:Number = ( sourisY - height/2);
```

```

        // déplace la balle avec un effet de ralentissement (inertie)
        x -= (x - destinationX)*.1;
        y -= (y - destinationY)*.1;

        if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY
    ) < 1 )
    {

        removeEventListener ( Event.ENTER_FRAME, mouvement );

        // diffusion de l'événement motionComplete
        dispatchEvent ( new Event ("mouvementTermine") );

    }
}

```

Notre symbole balle diffuse désormais un événement **mouvementTermine**. Il ne nous reste plus qu'à l'écouter :

```

// création du symbole
// affiche : [object Balle]
var maBalle:Balle = new Balle();

// ajout à la liste d'affichage
addChild ( maBalle );

// écoute de l'événement personnalisé motionComplete
maBalle.addEventListener ( "mouvementTermine", arrivee );

// fonction écouteur
function arrivee ( pEvt:Event ):void
{

    trace("mouvement terminé !");

}

```

La fonction écouteur **arrivee** est déclenchée lorsque la balle arrive à destination. En cas de réutilisation de la classe **Balle** nous savons que celle-ci diffuse l'événement **mouvementTermine**. Libre à nous de décider quoi faire lorsque l'événement est diffusé, la réutilisation de la classe **Balle** est donc facilitée.

Notre code fonctionne très bien, mais il n'est pas totalement optimisé. Voyez-vous ce qui pose problème ?

Souvenez-vous, lors du chapitre 3 intitulé *Le modèle événementiel*, nous avons vu que nous ne qualifions **jamais** le nom des événements directement. Cela rend notre code rigide et non standard.

Nous préférons l'utilisation de constantes de classe. Nous définissons donc une propriété constante **MOUVEMENT\_TERMINE** au sein de la classe **Balle** contenant le nom de l'événement diffusé :

```
// stocke le nom de l'événement diffusé
public static const MOUVEMENT_TERMINE:String = "mouvementTermine";
```

Puis nous ciblons la propriété afin d'écouter l'événement :

```
// écoute de l'événement personnalisé Balle.MOUVEMENT_TERMINE
maBalle.addEventListener ( Balle.MOUVEMENT_TERMINE, arrivee );
```

Nous modifions la méthode mouvement afin de cibler la constante

**Balle.MOUVEMENT\_TERMINE** :

```
private function mouvement ( pEvt:Event ):void
{
    // évalue la destination x et y
    var destinationX:Number = ( sourisX - width/2);
    var destinationY:Number = ( sourisY - height/2);

    // déplace la balle avec un effet de ralentissement (inertie)
    x -= (x - destinationX)*.1;
    y -= (y - destinationY)*.1;

    if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY
    ) < 1 )
    {
        removeEventListener ( Event.ENTER_FRAME, mouvement );

        // diffusion de l'événement motionComplete
        dispatchEvent ( new Event ( Balle.MOUVEMENT_TERMINE ) );
    }
}
```

Nous venons de traiter à travers cet exemple le cas le plus courant lors de la diffusion d'événements personnalisés. Voyons maintenant d'autres cas.

## A retenir

- Un événement doit porter un nom simple et intuitif.
- Afin de stocker le nom d'un événement nous utilisons **toujours** une propriété constante de classe.

## Etendre EventDispatcher

Rappelez-vous que toutes les classes résidant dans le paquetage **flash** héritent de la classe **EventDispatcher**. Dans vos développements, certaines classes n'hériteront pas de classes natives et n'auront pas la possibilité de diffuser des événements par défaut.



Prenons le cas d'une classe nommée `XMLLoader` devant charger des données XML. Nous souhaitons indiquer la fin du chargement des données en diffusant un événement approprié.

Le code suivant illustre le contenu de la classe :

```
package
{
    import flash.events.Event;

    public class XMLLoader
    {
        public function XMLLoader ()
        {
            // code non indiqué
        }

        public function charge ( pXML:String ):void
        {
            // code non indiqué
        }

        public function chargementTermine ( pEvt:Event ):void
        {
            dispatchEvent ( new Event ( XMLLoader.COMPLETE ) );
        }
    }
}
```

Si nous tentons de compiler cette classe, un message d'erreur nous avertira qu'aucune méthode `dispatchEvent` n'existe au sein de la classe. Afin d'obtenir les capacités de diffusion la classe `XMLLoader` hérite de la classe `EventDispatcher` :

```
package
{
    import flash.events.Event;
    import flash.events.EventDispatcher;

    public class XMLLoader extends EventDispatcher
    {
        public static const COMPLETE:String = "complete";
    }
}
```

```
public function XMLLoader ()
{
    // code non indiqué
}

public function charge ( pXML:String ):void
{
    // code non indiqué
}

public function chargementTermine ( pEvt:Event ):void
{
    dispatchEvent ( new Event ( XMLLoader.COMPLETE ) );
}
}
```

En sous classant `EventDispatcher`, la classe `XMLLoader` peut désormais diffuser des événements. Afin d'écouter l'événement `XMLLoader.COMPLETE`, nous écrivons le code suivant :

```
// création d'une instance de XMLLoader
var chargeurXML:XMLLoader = new XMLLoader();

// chargement des données
chargeurXML.charge ("news.xml");

// écoute de l'événement XMLLoader.COMPLETE
chargeurXML.addEventListener ( XMLLoader.COMPLETE, chargementTermine );

// fonction écouteur
function chargementTermine ( pEvt:Event ):void
{
    trace( "données chargées");
}
```

Lorsque nous appelons la méthode `charge`, les données XML sont chargées. Une fois le chargement terminé nous diffusons l'événement `XMLLoader.COMPLETE`.

Une partie du code de la classe `XMLLoader` n'est pas montré car nous n'avons pas encore traité la notion de chargement externe. Nous

reviendrons sur cette classe au cours du chapitre 14 intitulé *Chargement et envoi de données* afin de la compléter.

Bien que cette technique soit efficace, elle ne révèle pas être la plus optimisée. Comme nous l'avons vu lors du chapitre 8 intitulé *Programmation orientée objet*, ActionScript n'intègre pas d'héritage multiple. Ainsi, en héritant de la classe `EventDispatcher` la classe `XMLLoader` ne peut hériter d'une autre classe. Nous cassons la chaîne d'héritage. Pire encore, si la classe `XMLLoader` devait étendre une autre classe, nous ne pourrions étendre en même temps `EventDispatcher`.

Dans un scénario comme celui-ci nous allons utiliser une notion essentielle de la programmation orientée objet abordée au cours du chapitre 8. Peut être avez vous déjà une idée ?

## Stocker EventDispatcher

Afin de bien comprendre cette nouvelle notion, nous allons nous attarder sur la classe `Administrateur` développée lors du chapitre 8.

Voici le code de la classe `Administrateur` :

```
package
{
    // l'héritage est traduit par le mot clé extends
    public class Administrateur extends Joueur
    {
        public function Administrateur ( pPrenom:String, pNom:String,
        pAge:int, pVille:String )
        {
            super ( pPrenom, pNom, pAge, pVille );
        }

        // méthode permettant à l'administrateur de se présenter
        override public function sePresenter ( ):void
        {
            // déclenche la méthode surchargée
            super.sePresenter();

            trace("Je suis modérateur");
        }

        // méthode permettant de supprimer un joueur de la partie
        public function kickJoueur ( pJoueur:Joueur ):void
```

```
{  
  
    trace ("Kick " + pJoueur );  
  
}  
  
// méthode permettant de jouer un son  
public function jouerSon ( ):void  
  
{  
  
    trace("Joue un son");  
  
}  
  
}  
  
}
```

En découvrant la notion d'événements personnalisés nous décidons de diffuser un événement lorsqu'un joueur est supprimé de la partie. Il faudrait donc que la méthode `kickJoueur` puisse appeler la méthode `dispatchEvent`, ce qui est impossible pour le moment.

La classe `Administrateur` ne peut par défaut diffuser des événements, nous tentons donc d'étendre la classe `EventDispatcher`. Nous sommes alors confrontés à un problème, car la classe `Administrateur` étend déjà la classe `Joueur`.

Comment allons nous faire, sommes nous réellement bloqués ?

Souvenez vous, nous avons vu au cours du chapitre 8 une alternative à l'héritage. Cette technique décrivait une relation de type « possède un » au lieu d'une relation de type « est un ». En résumé, au lieu d'étendre une classe pour hériter de ses capacités nous allons créer une instance de celle-ci au sein de la classe et déléguer les fonctionnalités.

Ainsi au lieu de sous classer `EventDispatcher` nous allons stocker une instance de la classe `EventDispatcher` et déléguer la gestion des événements à celle-ci :

```
package  
  
{  
  
    import flash.events.EventDispatcher;  
  
    // l'héritage est traduit par le mot clé extends  
    public class Administrateur extends Joueur  
  
    {  
  
        // stocke l'instance d'EventDispatcher  
        private var diffuseur:EventDispatcher;
```

```

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )
        {
            super ( pPrenom, pNom, pAge, pVille );

            // création d'une instance d'EventDispatcher
            diffuseur = new EventDispatcher();
        }

        // méthode permettant à l'administrateur de se présenter
        override public function sePresenter ( ):void
        {
            // déclenche la méthode surchargée
            super.sePresenter();

            trace("Je suis modérateur");
        }

        // méthode permettant de supprimer un joueur de la partie
        public function kickJoueur ( pJoueur:Joueur ):void
        {
            // code gérant la déconnexion du joueur concerné
            trace ("Kick " + pJoueur );
        }

        // méthode permettant de jouer un son
        public function jouerSon ( ):void
        {
            trace("Joue un son");
        }
    }
}

```

Bien entendu, la classe `Administrateur` ne possède pas pour le moment les méthodes `dispatchEvent`, `addEventListener`, et `removeEventListener`.

C'est à nous de les implémenter, pour cela nous implémentons l'interface `flash.events.IEventDispatcher` :

```

package
{
    import flash.events.EventDispatcher;
    import flash.events.IEventDispatcher;
    import flash.events.Event;
}

```

```
// l'héritage est traduit par le mot clé extends
public class Administrateur extends Joueur implements IEventDispatcher

{

    // stocke l'instance d'EventDispatcher
    private var diffuseur:EventDispatcher;

    public function Administrateur ( pPrenom:String, pNom:String,
    pAge:int, pVille:String )

    {

        super ( pPrenom, pNom, pAge, pVille );

        // création d'une instance d'EventDispatcher
        diffuseur = new EventDispatcher();

    }

    // méthode permettant à l'administrateur de se présenter
    override public function sePresenter ( ):void

    {

        // déclenche la méthode surchargée
        super.sePresenter();

        trace("Je suis modérateur");

    }

    // méthode permettant de supprimer un joueur de la partie
    public function kickJoueur ( pJoueur:Joueur ):void

    {

        // code gérant la déconnexion du joueur concerné

        trace ("Kick " + pJoueur );

    }

    // méthode permettant de jouer un son
    public function jouerSon ( ):void

    {

        trace("Joue un son");

    }

    public function addEventListener( type:String, listener:Function,
    useCapture:Boolean=false, priority:int=0, useWeakReference:Boolean=false
    ):void

    {

        diffuseur.addEventListener( type, listener, useCapture, priority,
        useWeakReference );

    }

}
```

```
    }

    public function dispatchEvent( event:Event ):Boolean
    {
        return diffuseur.dispatchEvent( event );
    }

    public function hasEventListener( type:String ):Boolean
    {
        return diffuseur.hasEventListener( type );
    }

    public function removeEventListener( type:String, listener:Function,
useCapture:Boolean=false ):void
    {
        diffuseur.removeEventListener( type, listener, useCapture );
    }

    public function willTrigger( type:String ):Boolean
    {
        return diffuseur.willTrigger( type );
    }
}
}
```

Afin de rendre notre classe `Administrateur` diffuseur d'événements nous implémentons l'interface `IEventDispatcher`. Chacune des méthodes définissant le type `EventDispatcher` doivent donc être définies au sein de la classe `Administrateur`.

Nous définissons une constante de classe `KICK_JOUEUR`, stockant le nom de l'événement :

```
| public static const KICK_JOUEUR:String = "deconnecteJoueur";
```

Puis nous modifions la méthode `kickJoueur` afin de diffuser un événement `Administrateur.KICK_JOUEUR` :

```
| // méthode permettant de supprimer un joueur de la partie
| public function kickJoueur ( pJoueur:Joueur ):void
| {
|
|     // code gérant la déconnexion du joueur concerné
|
|     // diffuse l'événement Administrateur.KICK_JOUEUR
```

```
dispatchEvent ( new Event ( Administrateur.KICK_JOUEUR ) );  
}
```

Dans le code suivant, nous créons un modérateur et un joueur. Le joueur est supprimé de la partie, l'événement `Administrateur.KICK_JOUEUR` est bien diffusé :

```
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,  
"Los Angeles");  
  
// écoute l'événement Administrateur.KICK_JOUEUR  
monModo.addEventListener (Administrateur.KICK_JOUEUR, deconnexionJoueur );  
  
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");  
  
// un joueur est supprimé de la partie  
monModo.kickJoueur ( premierJoueur );  
  
// fonction écouteur  
function deconnexionJoueur ( pEvt:Event ):void  
{  
    trace( "Un joueur a quitté la partie !" );  
}
```

Grâce à la composition, nous avons pu rendre la classe `Administrateur` diffuseur d'événements. L'héritage n'est donc pas la seule alternative permettant de rendre une classe diffuseur d'événements.

## A retenir

- Lorsque nous ne pouvons pas étendre la classe `EventDispatcher` nous stockons une instance de celle-ci et déléguons les fonctionnalités.
- L'interface `IeventDispatcher` permet une implémentation obligatoire des différentes méthodes nécessaires à la diffusion d'événements.

## Passer des informations

La plupart des événements diffusés contiennent différentes informations relatives à l'état de l'objet diffusant l'événement. Nous avons vu lors du chapitre 6 intitulé *Intéractivité* que les classes `MouseEvent` ou `KeyboardEvent` possédaient des propriétés renseignant sur l'état de l'objet diffusant l'événement.

Dans l'exercice précédent, la classe `Administrateur` diffuse un événement `Administrateur.KICK_JOUEUR` mais celui-ci ne contient aucune information. Il serait intéressant que celui-ci nous renseigne sur le joueur supprimé. De la même manière la classe



`XMLLoader` pourrait diffuser un événement `XMLLoader.COMPLETE` contenant le flux XML chargé afin qu'il soit facilement utilisable par les écouteurs.

Afin de passer des informations lors de la diffusion d'un événement nous devons étendre la classe `Event` afin de créer un objet événementiel spécifique à l'événement diffusé. En réalité, nous nous plions au modèle défini par ActionScript 3. Lorsque nous écoutons un événement lié à la souris nous nous dirigeons instinctivement vers la classe `flash.events.MouseEvent`. De la même manière pour écouter des événements liés au clavier nous utilisons la classe `flash.events.KeyboardEvent`.

De manière générale, il convient de créer une classe événementielle pour chaque classe devant diffuser des événements contenant des informations particulières.

Nous allons donc étendre la classe `Event` et créer une classe nommée `AdministrateurEvent` :

```
package
{
    import flash.events.Event;

    public class AdministrateurEvent extends Event
    {
        public static const KICK_JOUEUR:String = "deconnecteJoueur";

        public function AdministrateurEvent ( type:String,
        bubbles:Boolean=false, cancelable:Boolean=false )
        {
            // initialisation du constructeur de la classe Event
            super( type, bubbles, cancelable );
        }

        // la méthode clone doit être surchargée
        public override function clone ():Event
        {
            return new AdministrateurEvent ( type, bubbles, cancelable )
        }

        // la méthode toString doit être surchargée
        public override function toString ():String
        {

```

```

        return '[AdministrateurEvent type="'+ type +'" bubbles=' +
bubbles + ' cancelable=' + cancelable + ']';
    }

}

}

```

Puis nous utilisons une instance de cette classe afin de diffuser l'événement :

```

// méthode permettant de supprimer un joueur de la partie
public function kickJoueur ( pJoueur:Joueur ):void

{

    // code gérant la déconnexion du joueur concerné

    // diffuse l'événement AdministrateurEvent.KICK_JOUEUR
    dispatchEvent ( new AdministrateurEvent (
AdministrateurEvent.KICK_JOUEUR ) );

}

```

Il est important de noter que jusqu'à présent nous n'avions diffusé des événements qu'avec la classe `Event`. Lorsque nous définissons une classe événementielle spécifique nous stockons la constante de classe dans celle-ci.

Ainsi au lieu de cibler le nom de l'événement sur la classe diffusant l'événement :

```

// écoute l'événement Administrateur.KICK_JOUEUR
monModo.addEventListener ( Administrateur.KICK_JOUEUR, joueurQuitte );

```

Nous préférons stocker le nom de l'événement au sein d'une constante de la classe événementielle :

```

// écoute l'événement AdministrateurEvent.KICK_JOUEUR
monModo.addEventListener ( AdministrateurEvent.KICK_JOUEUR, joueurQuitte );

```

A ce stade, aucune information ne transite par l'objet événementiel `AdministrateurEvent`. Afin de stocker des données supplémentaires nous définissons les propriétés voulues au sein de la classe puis nous affectons leur valeur selon les paramètres passés durant l'instanciation de l'objet événementiel :

```

package

{

    import flash.events.Event;

    public class AdministrateurEvent extends Event

    {

        public static const KICK_JOUEUR:String = "deconnecteJoueur";
        public var joueur:Joueur;
    }
}

```

```

        public function AdministrateurEvent ( type:String,
        bubbles:Boolean=false, cancelable:Boolean=false, pJoueur:Joueur=null )

        {

            // initialisation du constructeur de la classe Event
            super( type, bubbles, cancelable );

            // stocke le joueur supprimé
            joueur = pJoueur;

        }

        // la méthode clone doit être surchargée
        public override function clone ():Event

        {

            return new AdministrateurEvent ( type, bubbles, cancelable )

        }

        // la méthode toString doit être surchargée
        public override function toString ():String

        {

            return "[AdministrateurEvent type : " + type + ", bubbles : " +
            bubbles + ", cancelable : " + cancelable + "]";

        }

    }
}

```

Puis nous modifions la méthode `kickJoueur` de la classe `Administrateur` afin de passer en paramètre le joueur supprimé :

```

// méthode permettant de supprimer un joueur de la partie
public function kickJoueur ( pJoueur:Joueur ):void

{

    // code gérant la déconnexion du joueur concerné

    // diffuse l'événement Administrateur.KICK_JOUEUR
    dispatchEvent ( new AdministrateurEvent ( AdministrateurEvent.
    KICK_JOUEUR, false, false, pJoueur ) );

}

```

Lorsque l'événement est diffusé, la fonction écouteur accède à la propriété `joueur` afin de savoir quel joueur a quitté la partie :

```

var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// écoute l'événement Administrateur.KICK_JOUEUR
monModo.addEventListener ( AdministrateurEvent.KICK_JOUEUR, joueurQuitte );

var premierJoueur:Joueur = new Joueur ( "Bobby", "Womack", 66, "Detroit");

```

```
// un joueur est supprimé de la partie
monModo.kickJoueur ( premierJoueur );

// fonction écouteur
function joueurQuitte ( pEvt:AdministrateurEvent ):void
{
    // affiche : [AdministrateurEvent type="deconnecteJoueur" bubbles=false
cancelable=false]
    trace( pEvt );

    // affiche Bobby a quitté la partie !
    trace( pEvt.joueur.prenom + " a quitté la partie !");
}
```

En testant le code précédent, fonction écouteur `joueurQuitte` est notifié de l'événement `AdministrateurEvent.KICK_JOUEUR` et reçoit en paramètre un objet événementiel de type `AdministrateurEvent`.

La propriété `joueur` retourne le joueur supprimé de la partie, nous n'avons plus qu'à accéder aux propriétés voulues.

## A retenir

- Afin de passer des paramètres lors de la diffusion d'un événement, nous devons obligatoirement étendre la classe `Event`.
- Les classes et sous-classes d'`Event` représentent les objets événementiels diffusés.

## Menu et événement personnalisé

En fin de chapitre précédent nous avons développé un menu entièrement dynamique. Nous ne l'avons pas totalement finalisé car il nous manquait la notion d'événements personnalisés.

Souvenez-vous, nous avons besoin de diffuser un événement qui contiendrait le nom du SWF lié au bouton cliqué.

Au sein d'un répertoire `evenements` lui-même placé au sein du répertoire `org` nous créons une classe `ButtonEvent` :

```
package org.bytearray.evenements
{
    import flash.events.Event;

    public class ButtonEvent extends Event
    {
```

```
        public static const CLICK:String = "buttonClick";
        public var lien:String;

        public function ButtonEvent ( type:String, bubbles:Boolean=false,
cancelable:Boolean=false, pLien:String=null )

        {

            // initialisation du constructeur de la classe Event
            super( type, bubbles, cancelable );

            // stocke le lien lié au bouton cliqué
            lien = pLien;

        }

        // la méthode clone doit être surchargée
        public override function clone ():Event

        {

            return new ButtonEvent ( type, bubbles, cancelable, lien )

        }

        // la méthode toString doit être surchargée
        public override function toString ():String

        {

            return '[ButtonEvent type="'+ type +'" bubbles=' + bubbles + '
eventPhase=' + eventPhase + ' cancelable=' + cancelable + ']';

        }

    }
}
```

Dans le constructeur nous ajoutons la ligne suivante afin d'écouter l'événement `MouseEvent.CLICK` :

```
// écoute de l'événement MouseEvent.CLICK
addEventListener ( MouseEvent.CLICK, clicSouris );
```

Nous définissons la méthode écouteur `clicSouris` qui diffuse l'événement `ButtonEvent.CLICK` en passant le SWF correspondant :

```
private function clicSouris ( pEvt:MouseEvent ):void

{

    // diffusion de l'événement ButtonEvent.CLICK
    dispatchEvent ( new ButtonEvent ( ButtonEvent.CLICK, true, false, swf
) );

}
```

Puis nous écoutons l'événement `ButtonEvent.CLICK` auprès de chaque bouton, nous utilisons la phase de capture afin d'optimiser le code :

```
// import de la classe Bouton
import org.bytearray.ui.Button;
import org.bytearray.evenements.ButtonEvent;

// tableau associatif contenant les données
var donnees:Array = new Array();
donnees.push ( { legende : "Accueil", vitesse : 1, swf : "accueil.swf",
couleur : 0x999900 } );
donnees.push ( { legende : "Photos", vitesse : 1, swf : "photos.swf", couleur : 0x881122 } );
donnees.push ( { legende : "Blog", vitesse : 1, swf : "blog.swf", couleur : 0x995471 } );
donnees.push ( { legende : "Liens", vitesse : 1, swf : "liens.swf", couleur : 0xCC21FF } );
donnees.push ( { legende : "Forum", vitesse : 1, swf : "forum.swf", couleur : 0x977821 } );

// nombre de rubriques
var lng:int = donnees.length;

// conteneur du menu
var conteneurMenu:Sprite = new Sprite();

addChild ( conteneurMenu );

for (var i:int = 0; i< lng; i++ )
{

    // récupération des infos
    var legende:String = donnees[i].legende;
    var couleur:Number = donnees[i].couleur;
    var vitesse:Number = donnees[i].vitesse;
    var swf:String = donnees[i].swf;

    // création des boutons
    var monBouton:Button = new Button( 60, 120, swf, couleur, vitesse, legende );

    // positionnement
    monBouton.y = 50 * i;

    // ajout à la liste d'affichage
    conteneurMenu.addChild ( monBouton );

}

// écoute de l'événement ButtonEvent.CLICK pour la phase de capture
conteneurMenu.addEventListener ( ButtonEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:ButtonEvent ):void
{

    // affiche :
    /*accueil.swf
    photos.swf
```

```

        blog.swf
        liens.swf
        */
        trace( pEvt.lien );
    }

```

Lorsque l'événement `MouseEvent.CLICK` est diffusé, la fonction `clicBouton` est déclenchée. La propriété `lien` de l'objet événementiel de type `MouseEvent` permet de charger le SWF correspondant. Celle-ci pourrait contenir dans une autre application une URL à atteindre lorsque l'utilisateur clique sur un bouton.

Il serait tout à fait envisageable de définir de nouvelles propriétés au sein de la classe `MouseEvent` telles `legende`, `couleur` ou `vitesse` ou autres afin de passer les caractéristiques de chaque bouton.

Voici le code complet de la classe `Button` :

```

package org.bytearray.ui
{
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.text.Font;
    import flash.text.TextFormat;
    import org.events.MouseEvent;
    // import des classes liées Tween au mouvement
    import fl.transitions.Tween;
    import fl.transitions.easing.Bounce;
    // import de la classe MouseEvent
    import flash.events.MouseEvent;
    // import de la classe TextField et TextFieldAutoSize
    import flash.text.TextField;
    import flash.text.TextFieldAutoSize;

    public class Button extends Sprite
    {
        // stocke le fond du bouton
        private var fondBouton:Shape;
        // stocke l'objet Tween pour les différents état du bouton
        private var etatTween:Tween;
        // stocke les références aux boutons
        private static var tableauBoutons:Array = new Array();
        // stocke la couleur en cours du bouton
        private var couleur:Number;
        // stocke la vitesse d'ouverture de chaque bouton
        private var vitesse:Number;
        // légende du bouton
        private var legende:TextField;
        // formatage des légendes
        private var formatage:TextFormat;
        // swf associé
        private var swf:String;
    }
}

```

```
public function Button ( pWidth:Number, pHeight:Number, pSWF:String,
pCouleur:Number, pVitesse:Number, pLegende:String )

{

    // ajoute chaque instance au tableau
    Button.tableauBoutons.push ( this );

    // création du fond du bouton
    fondBouton = new Shape();

    // ajout à la liste d'affichage
    addChild ( fondBouton );

    // crée le champ texte
    legende = new TextField();

    // redimensionnement automatique du champ texte
    legende.autoSize = TextFieldAutoSize.LEFT;

    // ajout à la liste d'affichage
    addChild ( legende );

    // affecte la légende
    legende.text = pLegende;

    // active l'utilisation de police embarquée
    legende.embedFonts = true;

    // crée un objet de formatage
    formatage = new TextFormat();

    // taille de la police
    formatage.size = 12;

    // instanciation de la police embarquée
    var police:MaPolice = new MaPolice();

    // affectation de la police au formatage
    formatage.font = police.fontName;

    // affectation du formatage au champ texte
    legende.setTextFormat ( formatage );

    // rend le champ texte non sélectionnable
    legende.selectable = false;

    // stocke la couleur passée en paramètre
    couleur = pCouleur;

    // stocke le nom du SWF
    swf = pSWF;

    // dessine le bouton
    fondBouton.graphics.beginFill ( couleur, 1 );
    fondBouton.graphics.drawRect ( 0, 0, pWidth, pHeight );

    // activation du mode bouton
    buttonMode = true;

    // désactivation des objets enfants
    mouseChildren = false;
```



```

        // affecte la vitesse passée en paramètre
        setVitesse ( pVitesse );

        // création de l'objet Tween
        etatTween = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

        // écoute de l'événement MouseEvent.ROLL_OVER
        addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

        // écoute de l'événement MouseEvent.CLICK
        addEventListener ( MouseEvent.CLICK, clicSouris );

    }

    private function clicSouris ( pEvt:MouseEvent ):void

    {

        // diffusion de l'événement ButtonEvent.CLICK
        dispatchEvent ( new ButtonEvent ( ButtonEvent.CLICK, true, false,
swf ) );

    }

    // déclenché lors du clic sur le bouton
    private function survolSouris ( pEvt:MouseEvent ):void

    {

        // stocke la longueur du tableau
        var lng:int = Button.tableauBoutons.length;

        for (var i:int = 0; i<lng; i++ )
Button.tableauBoutons[i].fermer();

        // démarrage de l'animation
        etatTween.continueTo ( 2, vitesse );

    }

    // méthode permettant de refermer le bouton
    private function fermer ():void

    {

        // referme le bouton
        etatTween.continueTo ( 1, vitesse );

    }

    // gère l'affectation de la vitesse
    public function setVitesse ( pVitesse:Number ):void

    {

        // affecte la vitesse
        if ( pVitesse >= 1 && pVitesse <= 10 ) vitesse = pVitesse;

        else

        {

```

```
        trace("Erreur : Vitesse non correcte, la valeur doit être  
comprise entre 1 et 10");  
        vitesse = 1;  
    }  
}  
}  
}
```

La diffusion d'événements personnalisés est un point essentiel dans tout développement orienté objet ActionScript. En diffusant nos propres événements nous rendons nos objets compatibles avec le modèle événementiel ActionScript 3 et facilement réutilisables.

En livrant une classe à un développeur tiers, celui-ci regardera en premier lieu les capacités offertes par celle-ci puis s'attardera sur les différents événements diffusés pour voir comment dialoguer facilement avec celle-ci.

## A retenir

- Nous pouvons définir autant de propriétés que nous le souhaitons au sein de l'objet événementiel diffusé.

Nous allons nous intéresser maintenant à la notion de classe du document. Cette nouveauté apportée par Flash CS3 va nous permettre de rendre nos développements ActionScript 3 plus aboutis.

En avant pour le chapitre 11 intitulé *Classe du document*.

# 11

## Classe du document

<b>INTERETS .....</b>	<b>1</b>
<b>LA CLASSE MAINTIMELINE .....</b>	<b>2</b>
<b>CLASSE DU DOCUMENT.....</b>	<b>3</b>
LIMITATIONS DE LA CLASSE SPRITE .....	6
ACTIONS D'IMAGES.....	8
<b>DECLARATION AUTOMATIQUE DES OCCURRENCES.....</b>	<b>9</b>
DECLARATION MANUELLE DES OCCURRENCES .....	11
<b>AJOUTER DES FONCTIONNALITES .....</b>	<b>12</b>
<b>INITIALISATION DE L'APPLICATION.....</b>	<b>15</b>
ACCÈS GLOBAL À L'OBJET STAGE.....	18
AUTOMATISER L'ACCÈS GLOBAL À L'OBJET STAGE .....	21

### Intérêts

Depuis l'introduction d'ActionScript 2, les développeurs avaient pour habitude de créer une classe servant de point d'entrée à leur application. Celle-ci instanciat d'autres objets ayant généralement besoin d'accéder au scénario principal. Il fallait donc passer une référence au scénario principal.

Il n'était donc pas rare de trouver sur une image du scénario de l'application le code suivant :

```
var monApplication:Application = new Application ( this );
```

L'application pouvait aussi être initialisée à l'aide d'une simple méthode statique :

```
Application.initialise ( this );
```

D'autres développeurs utilisaient une autre technique, consistant à changer le contexte d'exécution de la classe principale avec le code suivant :

```
this.__proto__ = Application.prototype;
Application['apply']( this , null );
```

En utilisant le mot-clé `this` au sein de l'instance de la classe `Application` nous faisons alors référence au scénario principal.

ActionScript 3 intègre une nouvelle fonctionnalité appelée *Classe du document* permettant d'éviter d'avoir recours à ces différentes astuces.

## La classe MainTimeline

Comme nous l'avons vu lors du chapitre 4 intitulé *La liste d'affichage* le lecteur Flash est constitué d'un objet `Stage` situé au sommet de la liste d'affichage. Lorsqu'un SWF est lu dans le lecteur, celui-ci ajoute le scénario principal du SWF chargé en tant qu'enfant de l'objet `Stage` :

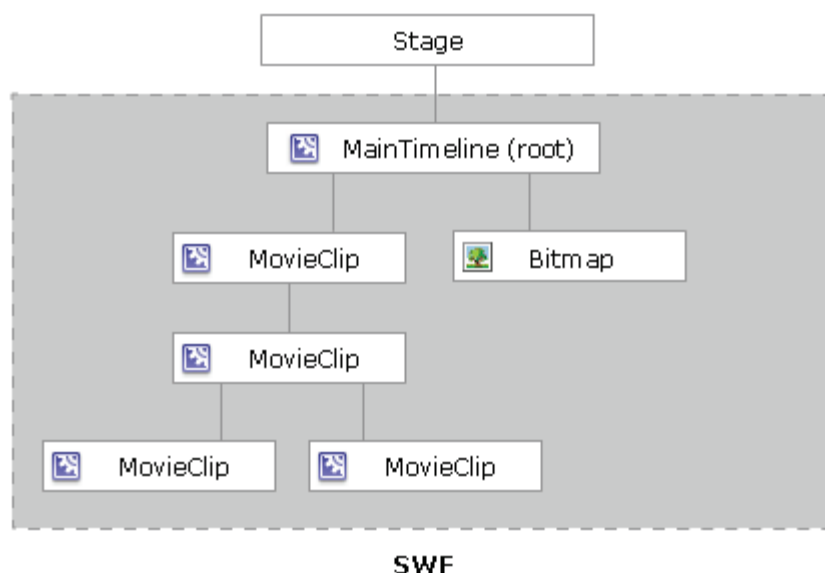


Figure 11-1. Classe du document *MainTimeline*.

Par défaut le scénario d'un SWF est représenté par la classe `MainTimeline`. Nous pouvons nous en rendre compte très facilement. Dans un nouveau document Flash CS3, testez le code suivant sur le scénario principal :

```
// affiche : [object MainTimeline]
trace( this );
```

Nous allons voir que nous ne sommes pas limités à cette classe.

## A retenir

- Par défaut le scénario principal est représenté par la classe `MainTimeline`.

## Classe du document

Flash CS3 intègre une nouvelle fonctionnalité permettant d'utiliser une instance de sous classe graphique comme scénario principal. Attention, celle-ci doit hériter obligatoirement d'une classe graphique telle `flash.display.Sprite` ou `flash.display.MovieClip`.

Il est techniquement possible d'utiliser une sous classe de `flash.display.Shape` mais il serait impossible d'ajouter un objet graphique sur le scénario principal, la classe `Shape` n'étant pas une sous classe de `flash.display.DisplayObjectContainer`.

Si nous utilisons comme classe du document une sous-classe graphique nommée `Application`, Flash ajoute aussitôt une instance de celle-ci comme scénario principal.

La figure 11-2 illustre l'idée :

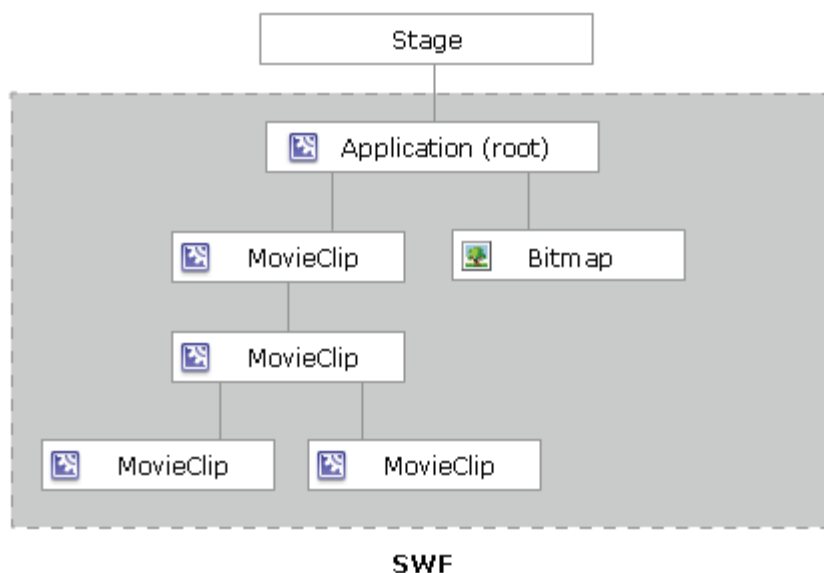


Figure 11-2. Classe du document `Application`.

Si nous ne spécifions aucune classe, le lecteur crée une instance de `MainTimeline`.

Nous allons créer un nouveau document Flash CS3 puis créer à côté un répertoire `org` dans lequel nous créons un répertoire `document`.

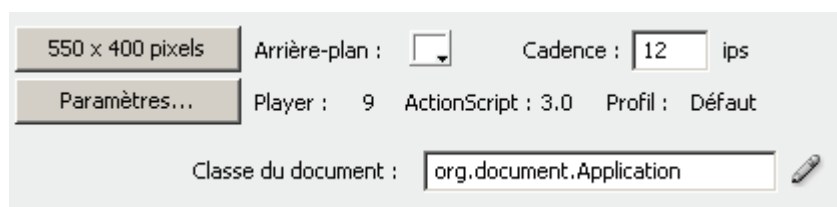
Au sein du répertoire `document` nous définissons une classe `Application` dont voici le code :

```
package org.bytearray.document
{
    import flash.display.Sprite;

    public class Application extends Sprite
    {
        public function Application ()
        {
            // affiche : [object Application]
            trace( this );
        }
    }
}
```

Grâce à la notion de classe du document, le mot-clé `this` fait ici référence à l'instance de la classe `Application`, donc au scénario principal.

Afin de lier cette classe du document à notre document Flash nous utilisons le champ *Classe du document* du panneau *Inspecteur de propriétés* illustré en figure 11-3 :



*Figure 11-3. Champ Classe du document.*

Nous spécifions le packaging complet de la classe `Application`.

Il est intéressant de noter que l'instanciation de la classe `Application` est transparente. Lorsque l'animation est chargée, le lecteur Flash crée automatiquement une instance de la classe `Application` et l'ajoute en tant qu'enfant de l'objet `Stage` :

```
package org.bytearray.document
{
```

```
import flash.display.Sprite;

public class Application extends Sprite
{
    public function Application ()
    {
        // affiche : [object Stage]
        trace( parent );

        // affiche : [object Stage]
        trace( stage );

        // affiche : [object Application]
        trace( this );
    }
}
```

Si nous écoutons l'événement `Event.ADDED_TO_STAGE` nous remarquons que celui-ci est bien diffusé :

```
package org.bytearray.document
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Application extends Sprite
    {
        public function Application ()
        {
            // écoute de l'événement Event.ADDED_TO_STAGE
            // est diffusé automatiquement car le lecteur
            // ajoute à la liste d'affichage une instance
            // de la classe Application
            addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );
        }

        private function ajoutAffichage ( pEvt:Event ):void
        {
            // affiche : [Event type="addedToStage" bubbles=true
            // cancelable=false eventPhase=2]
            trace( pEvt );
        }
    }
}
```

```
    }  
}
```

Il faut donc considérer la classe du document comme le point d'entrée de notre projet. Nous avons dans notre exemple utilisé une sous-classe de `Sprite` comme classe du document. Nous allons voir dans quelle mesure son utilisation est limitée dans ce contexte.

## A retenir

- Pour affecter une classe du document, nous utilisons le champ *Classe du document* de l'inspecteur de propriétés.
- L'instanciation de classe du document est transparente, le lecteur s'en charge automatiquement.
- L'instance de la classe du document devient le scénario principal.

## Limitations de la classe `Sprite`

Rappelez-vous, lors du chapitre 4 intitulé *La liste d'affichage* nous avons découvert que la classe `Sprite` ne disposait pas de scénario.

Ainsi, lorsque la classe du document est une sous-classe de `Sprite` il est impossible de faire appel aux différentes méthodes de manipulation du scénario telles `gotoAndPlay`, `gotoAndStop` ou autres.

Le code suivant ne peut être compilé :

```
package org.bytearray.document  
{  
    import flash.display.Sprite;  
    public class Application extends Sprite  
    {  
        public function Application ()  
        {  
            gotoAndStop ( "intro" );  
        }  
    }  
}
```

L'erreur suivante est générée à la compilation :

```
1180: Appel à une méthode qui ne semble pas définie, gotoAndStop.
```



Nous utiliserons donc une sous-classe de `Sprite` comme classe du document lorsque nous n’aurons pas besoin de scénario.

Nous pourrions penser qu’il s’agit de la seule limitation de la classe `Sprite`, mais il est aussi impossible d’ajouter du code sur les images du scénario. Les lignes suivantes sont posées sur l’image 1 de notre animation :

```
var monClip:MovieClip = new MovieClip();
```

L’erreur suivante est générée :

```
1180: Appel à une méthode qui ne semble pas définie, addFrameScript.
```

Une simple ligne de code commentée empêche la compilation :

```
//var monClip:MovieClip = new MovieClip();
```

La puissance de la classe du document réside dans l’interaction entre le code des images et les fonctionnalités apportées par la classe du document. L’utilisation d’une sous-classe de `Sprite` comme classe du document est donc généralement déconseillée.

En étendant la classe `MovieClip`, nous pouvons ajouter des actions d’images et manipuler le scénario.

Nous modifions la classe `Application` afin d’obtenir le code suivant :

```
package org.bytearray.document
{
    import flash.display.MovieClip;

    public class Application extends MovieClip
    {
        public function Application ()
        {

        }

    }
}
```

Dans le cas d’une application nécessitant un minimum d’animations et de code sur les images nous préférons étendre la classe `MovieClip`.

## Actions d'images

Toute action d'image s'exécute dans le contexte de l'instance de la classe du document. Prenons un exemple simple, nous ajoutons une méthode `afficheMenu` au sein de la classe `Application` :

```
package org.bytearray.document
{
    import flash.display.MovieClip;

    public class Application extends MovieClip
    {
        public function Application ()
        {

        }

        // méthode de création du menu
        public function afficheMenu ( ):void
        {

            trace("création du menu");

        }

    }
}
```

La méthode `afficheMenu` devient donc une méthode du scénario principal, sur n'importe quelle image nous pouvons l'exécuter en plaçant le code suivant :

```
// stop le scénario principal
stop();

// appelle la méthode afficheMenu de la classe du document
// affiche : création du menu
afficheMenu();
```

Nous stoppons le scénario principal puis appelons la méthode `afficheMenu` définie au sein de la classe du document.

Cette capacité à pouvoir déclencher différentes fonctionnalités de la classe du document depuis les images rend le développement plus souple. Lorsqu'un simple effet ou un ensemble de mécanismes complexes doivent être déclenchés nous l'appelons directement depuis le scénario.

---

## A retenir

---

- L'utilisation d'une sous classe de `Sprite` comme classe du document est généralement déconseillée.
- Nous préférons généralement utiliser une sous-classe de `MovieClip`.
- La puissance réside dans l'interaction entre les actions d'images et les fonctionnalités de la classe du document.

## Déclaration automatique des occurrences

Afin d'accéder aux objets graphiques depuis la classe du document nous utilisons deux techniques.

Comme nous l'avons vu au cours du chapitre 9 intitulé *Etendre les classes natives*, l'option *Déclarer automatiquement les occurrences de scène* est activée par défaut dans Flash CS3.

Dans l'exemple suivant nous créons un symbole clip auquel nous associons une classe `Personnage`, puis nous posons une occurrence sur la scène, nommée `monPersonnage` :

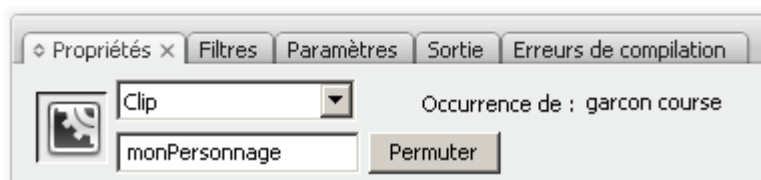


Figure 11-4. Occurrence `monPersonnage`.

A la compilation, une propriété `monPersonnage` est automatiquement ajoutée et référence l'occurrence :

```
package org.bytearray.document
{
    import flash.display.MovieClip;

    public class Application extends MovieClip
```

```
{  
  
    public function Application ()  
    {  
  
        // affiche : [object Personnage]  
        trace( monPersonnage );  
  
    }  
  
}
```

On pourrait également recourir à la méthode `getChildByName` :

```
package org.bytearray.document  
{  
  
    import flash.display.MovieClip;  
  
    public class Application extends MovieClip  
    {  
  
        public function Application ()  
        {  
  
            // suppression du symbole monPersonnage  
            removeChild ( getChildByName ( "monPersonnage" ) );  
  
        }  
  
    }  
  
}
```

Le code suivant supprime l'occurrence `monPersonnage` posée sur le scénario principal :

```
package org.bytearray.document  
{  
  
    import flash.display.MovieClip;  
  
    public class Application extends MovieClip  
    {  
  
        public function Application ()  
        {  
  
            // suppression du symbole monPersonnage  
            removeChild ( monPersonnage );  
  
        }  
  
    }  
  
}
```

```
}
```

```
}
```

Lorsque l'option *Déclarer automatiquement les occurrences de scène* est activée, le compilateur ajoute automatiquement au sein de la classe conteneur des propriétés pointant vers les occurrences créées depuis l'environnement auteur.

Cette déclaration automatique des occurrences empêche le développeur de voir quels sont les objets utilisés au sein de la classe et ne facilite pas le déboguage de l'application. C'est pour cette raison que nous préférons généralement désactiver la déclaration automatique dans un projet géré par des développeurs seulement.

A l'inverse, dans un contexte d'interactions développeurs-designers le graphiste peut être amené à poser un nouvel objet sur la scène et lui donner un nom d'occurrence afin d'enrichir graphiquement l'application.

Si l'option *Déclarer automatiquement les occurrences de scène* est désactivée, le graphiste, ne pourra plus compiler le projet sans que le développeur n'ajoute une propriété au sein de la classe du document correspondant à l'occurrence ajoutée. Cela risque donc de gêner le flux de travail au sein d'une équipe.

## Déclaration manuelle des occurrences

En désactivant la déclaration automatique des occurrences, nous devons définir manuellement au sein de la classe des propriétés du même nom que les occurrences :

```
package org.bytearray.document

{

    import flash.display.MovieClip;

    public class Application extends MovieClip

    {

        // propriété liée à l'occurrence
        public var monPersonnage:MovieClip

        public function Application ()

        {

            // suppression du symbole monPersonnage
            removeChild ( monPersonnage );

        }

    }

}
```

```
}  
}
```

La propriété définie manuellement doit obligatoirement être publique. Si nous la rendons privée à l'aide de l'attribut `private` le compilateur génère une erreur :

```
ReferenceError: Error #1056: Impossible de créer la propriété monPersonnage sur  
org.bytearray.document.Application.
```

Lorsque le compilateur tente d'affecter la propriété `monPersonnage` afin de la faire pointer vers l'occurrence, celle-ci est privée et n'est donc pas accessible depuis l'extérieur de la classe. Ainsi, les propriétés liées aux occurrences doivent **toujours** être publique.

Si nous définissons la propriété et que l'option *Déclarer les occurrences de scène* est activée, une erreur à la compilation s'affiche :

```
1151: Conflit dans la définition monPersonnage dans l'espace de nom internal.
```

Le compilateur tente de définir une propriété pointant vers l'occurrence posée sur la scène, mais celui-ci rencontre une propriété du même nom déjà définie. Un conflit de propriétés est généré.

### A retenir

- De manière générale, il est préférable de désactiver la déclaration automatique des occurrences de scène.
- Dans un contexte d'interactions designers-développeurs, il est préférable d'activer la déclaration automatique des occurrences.
- Les propriétés liées aux occurrences doivent toujours être publique.

## Ajouter des fonctionnalités

Il faut bien comprendre que lorsqu'une classe du document est définie, son instance représente le scénario principal.

Nous avons découvert lors du chapitre 4 intitulé *La liste d'affichage* différents comportements liés à l'accès aux objets graphiques. Souvenez-vous en : en ActionScript 2 nous pouvions écrire le code suivant :

```
gotoAndStop (20);  
monPersonnage._alpha = 100;
```

Même si l'occurrence `monPersonnage` n'était présente qu'à l'image 20, nous pouvions depuis n'importe quelle image y accéder. Cela n'est plus vrai en ActionScript 3.

Afin d'accéder correctement à l'objet `monPersonnage` nous pouvons utiliser l'événement `Event.RENDER`. Cet événement est déclenché lorsque le lecteur a fini de rendre les données vectorielles. Afin qu'il soit diffusé nous devons appeler la méthode `invalidate` de la classe `Stage`.

Ainsi, nous pourrions accéder à l'objet `monPersonnage` avec le code suivant :

```
// écoute de l'événement Event.RENDER
addEventListener ( Event.RENDER, miseAJour );

function miseAJour ( pEvt:Event ):void
{
    monPersonnage.alpha = 1;

    // supprime l'écoute
    removeEventListener( Event.RENDER, miseAJour );
}

// déplace le tête de lecture
gotoAndStop (20);

// force le rafraîchissement
stage.invalidate();
```

La fonction `miseAJour` est déclenchée lorsque le lecteur a terminé d'afficher les objets graphiques présents à l'image 20. Nous pouvons donc accéder sans problème à l'occurrence `monPersonnage`.

Grâce à la classe du document, nous allons mettre en place une fonctionnalité rendant tout ce traitement transparent.

Au sein de la classe `Application` nous définissons une méthode `myGotoAndStop` :

```
package org.bytearray.document
{
    import flash.display.MovieClip;
    import flash.events.Event;

    public class Application extends MovieClip
    {
        // propriétés liées à l'occurrence
        public var monPersonnage:MovieClip
        // propriété permettant l'exécution de la fonction de rappel
        private var rappel:Function;

        public function Application ()
```

```
{  
}  
  
// méthode de déplacement de la tête de lecture personnalisé  
public function myGotoAndStop ( pImage:int, pFonction:Function ):void  
{  
  
    // écoute de l'événement Event.RENDER  
    addEventListener ( Event.RENDER, miseAJour );  
  
    // déplacement de la tête de lecture  
    gotoAndStop ( pImage );  
    // retourne un objet permettant  
    rappel = pFonction;  
  
    // force la diffusion de l'événement Event.RENDER  
    stage.invalidate();  
  
}  
  
private function miseAJour ( pEvt:Event ):void  
{  
  
    // nous tentons d'appeler la fonction de rappel  
    try  
    {  
  
        rappel();  
  
        // si cela échoue, nous affichons un message d'erreur  
    } catch ( pErreur:Error )  
    {  
  
        trace("Erreur : La méthode de rappel n'a pas été définie");  
  
        // dans tout les cas, nous supprimons l'écoute de l'événement  
Event.RENDER  
    } finally  
    {  
  
        removeEventListener ( Event.RENDER, miseAJour );  
  
    }  
  
}  
  
}
```

Ainsi lorsque nous souhaitons retrouver le comportement de la méthode `gotoAndStop` des anciennes versions d'ActionScript nous écrivons le code suivant :

```
// déplace la tête de lecture en image 10  
// lorsque tout les objets sont disponibles  
// la fonction actifAccessible est déclenchée  
myGotoAndStop ( 10, actifAccessible );
```



```
function actifAccessible ():void
{
    // affiche : [object Personnage]
    trace(monPersonnage);

    // affecte la rotation de l'occurrence
    monPersonnage.rotation = 90;
}

stop();
```

La fonction `actifAccessible` est déclenchée automatiquement lorsque les occurrences de l'image 20 sont disponibles.

Grâce au bloc `try, catch, finally` nous gérons les erreurs à l'exécution. Au cas où l'utilisateur oublierait de définir la fonction associée :

```
// déplace la tête de lecture en image 10
// lorsque tout les objets sont disponibles
// la fonction actifAccessible est déclenchée
// affiche : Erreur : La méthode de rappel n'a pas été définie
myGotoAndStop ( 10, actifAccessible );

// fonction non définie
var actifAccessible:Function;

stop();
```

L'erreur à l'exécution est gérée et nous affichons le message indiquant l'oubli de définition.

Grâce à la classe du document nous pouvons ajouter des fonctionnalités au scénario principal ou modifier certains comportements. Nous étendons les capacités du scénario en définissant de nouvelles méthodes au sein de la classe du document.

## A retenir

- Il est préférable de désactiver la déclaration automatique des occurrences.
- Les propriétés liées aux occurrences doivent toujours être publique.

## Initialisation de l'application

Nous allons supprimer l'occurrence nommée `monPersonnage` sur la scène, puis initialiser l'application en instanciant par programmation l'instance de `Personnage`.

Afin d'initialiser notre application, nous utilisons simplement le constructeur de la classe du document. Dans le code suivant nous instancions puis affichons le symbole **Personnage** :

```
package org.bytearray.document

{

    import flash.display.MovieClip;
    import flash.events.Event;

    public class Application extends MovieClip

    {

        // propriété permettant l'exécution de la fonction de rappel
        private var rappel:Function;

        public function Application ()

        {

            // création du symbole Personnage
            var autrePersonnage:Personnage = new Personnage();

            // ajout à la liste d'affichage
            addChild ( autrePersonnage );

            // l'occurrence est centrée
            autrePersonnage.x = (stage.stageWidth - autrePersonnage.width)/2;
            autrePersonnage.y = (stage.stageHeight -
autrePersonnage.height)/2;

        }

        // méthode de déplacement de la tête de lecture personnalisé
        public function myGotoAndStop ( pImage:int, pFonction:Function ):void

        {

            // écoute de l'événement Event.RENDER
            addEventListener ( Event.RENDER, miseAJour );

            // déplacement de la tête de lecture
            gotoAndStop ( pImage );

            // retourne un objet permettant
            rappel = pFonction;

            // force la diffusion de l'événement Event.RENDER
            stage.invalidate();

        }

        private function miseAJour ( pEvt:Event ):void

        {

            // nous tentons d'appeler la fonction de rappel
            try
            {
```

```
        rappel();

        // si cela échoue, nous affichons un message d'erreur
    } catch ( pErreur:Error )
    {

        trace("Erreur : La méthode de rappel n'a pas été définie");

        // dans tout les cas, nous supprimons
        // l'écoute de l'événement Event.RENDER
    } finally
    {

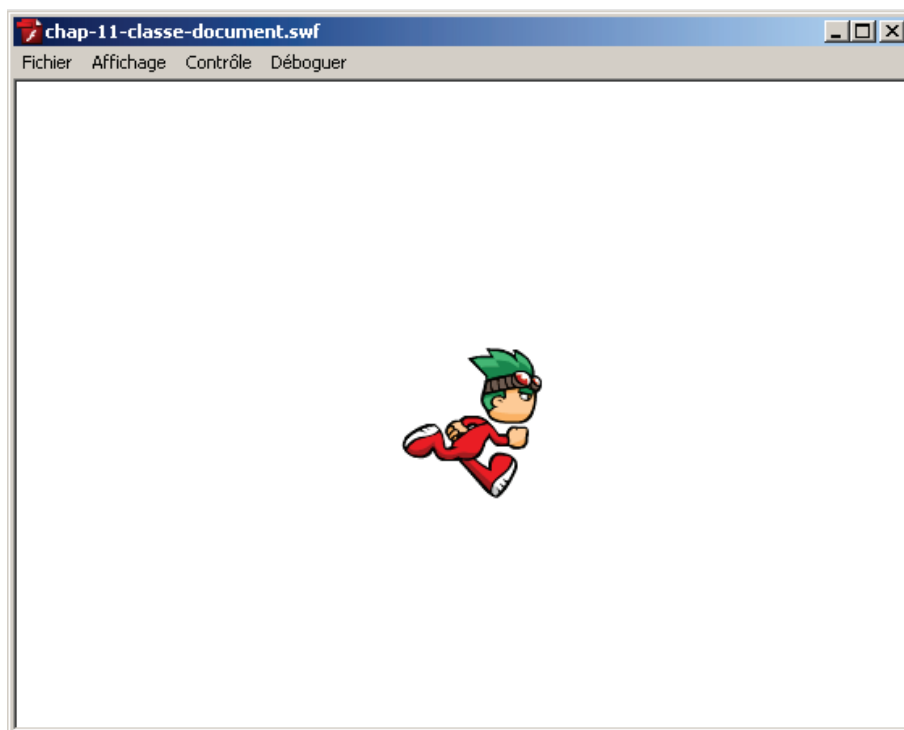
        removeEventListener ( Event.RENDER, miseAJour );

    }

}

}
```

La figure 11-6 illustre le résultat :



*Figure 11-6. Symbole **Personnage** affiché.*

Ne remarquez vous pas quelque chose de particulier ?

Nous devrions normalement écouter l'événement `Event.ADDED_TO_STAGE` pour pouvoir accéder à l'objet `Stage` de manière sécurisée. Dans le cas de la classe du document, le code est

exécuté une fois l'instance de la classe du document ajoutée à la liste d'affichage, ce qui est automatique et assuré par le lecteur. L'objet `Stage` est donc directement accessible par la propriété `stage` depuis le constructeur.

## Accès global à l'objet Stage

Il serait intéressant d'utiliser la classe du document comme point d'accès global à l'objet `Stage`. Pour cela, nous définissons une propriété statique `GLOBAL_STAGE` au sein de la classe `Application` :

```
// point d'accès à l'objet Stage
public static var GLOBAL_STAGE:Stage;
```

Puis nous modifions le constructeur :

```
public function Application ()
{
    // affecte une référence à l'objet Stage
    Application.GLOBAL_STAGE = stage;
}
```

En ciblant la propriété `Application.GLOBAL_STAGE`, n'importe quel objet obtiendra une référence à l'objet `Stage`.

Afin d'illustrer cette fonctionnalité, nous allons définir au sein d'un répertoire `ui` une classe `Fenetre` :

```
package org.bytearray.ui
{
    import flash.display.Shape;

    public class Fenetre extends Shape
    {
        public function Fenetre ( pLargeur:Number, pHauteur:Number,
        pRayon:Number, pCouleur:Number )
        {
            graphics.beginFill ( pCouleur );
            graphics.drawRoundRect ( 0, 0, pLargeur, pHauteur, pRayon );
        }
    }
}
```

Cette classe crée une forme rectangulaire illustrant une simple fenêtre.

En pointant vers la propriété `Application.GLOBAL_STAGE` nous obtenons un point d'accès global à l'objet `Stage` depuis n'importe quelle instance de la classe `Fenetre` :

```
package org.bytearray.ui

{

    import flash.display.Shape;
    import org.bytearray.document.Application;

    public class Fenetre extends Shape

    {

        public function Fenetre ( pLargeur:Number, pHauteur:Number,
        pRayon:Number, pCouleur:Number )

        {

            graphics.beginFill ( pCouleur );
            graphics.drawRoundRect ( 0, 0, pLargeur, pHauteur, pRayon );

            // nous centrons la fenêtre
            // en utilisant la propriété statique Application.GLOBAL_STAGE
            nous bénéficions d'un point d'accès global à l'objet Stage
            x = (Application.GLOBAL_STAGE.stageWidth - width)/2;
            y = (Application.GLOBAL_STAGE.stageHeight - height)/2;

        }

    }

}
```

Puis nous affichons la fenêtre :

```
package org.bytearray.document

{

    import flash.display.MovieClip;
    import flash.events.Event;
    import org.bytearray.ui.Fenetre;

    public class Application extends MovieClip

    {

        // point d'accès à l'objet Stage
        public static var GLOBAL_STAGE:Stage;
        // propriété permettant l'exécution de la fonction de rappel
        private var rappel:Function;

        public function Application ()

        {

            addEventListener ( Event.ADDED_TO_STAGE, activation );

        }

    }

}
```

```

private function activation ( pEvt:Event ):void
{
    // affecte une référence à l'objet Stage
    Application.GLOBAL_STAGE = stage;

    // création de la fenêtre
    var maFenetre:Fenetre = new Fenetre( 250, 250, 8, 0x88CCAA );

    // ajout à la liste d'affichage
    addChild ( maFenetre );
}

// méthode de déplacement de la tête de lecture personnalisé
public function myGotoAndStop ( pImage:int, pFonction:Function ):void
{
    // écoute de l'événement Event.RENDER
    addEventListener ( Event.RENDER, miseAJour );
    // déplacement de la tête de lecture
    gotoAndStop ( pImage );
    // retourne un objet permettant
    rappel = pFonction;
    // force la diffusion de l'événement Event.RENDER
    stage.invalidate();
}

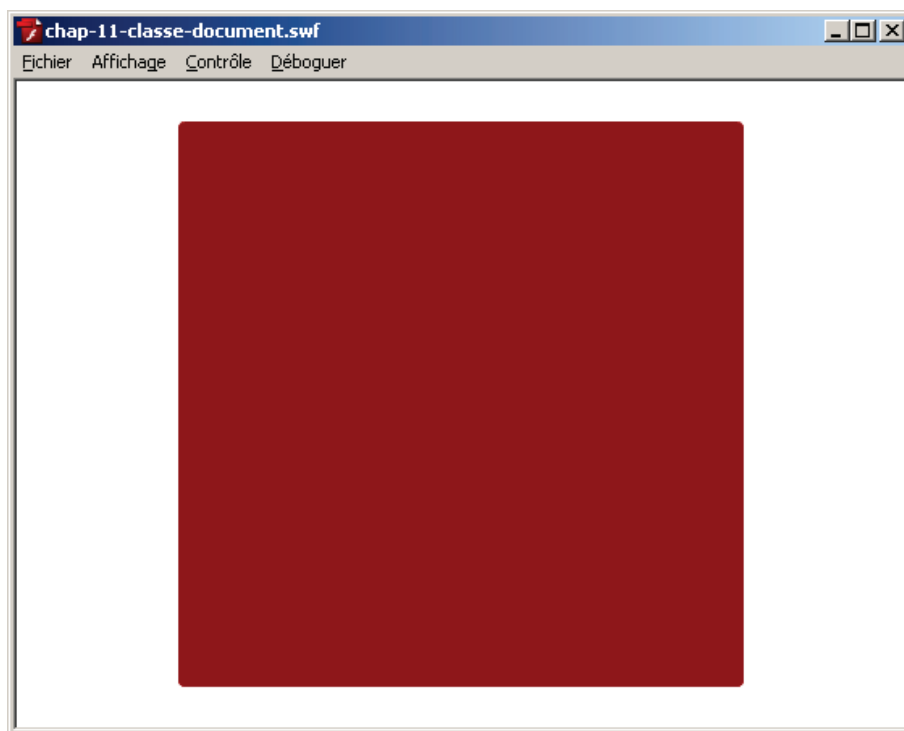
private function miseAJour ( pEvt:Event ):void
{
    // nous tentons d'appeler la fonction de rappel
    try
    {
        rappel();

        // si cela échoue, nous affichons un message d'erreur
    } catch ( pErreur:Error )
    {
        trace("Erreur : La méthode de rappel n'a pas été définie");

        // dans tout les cas, nous supprimons l'écoute de l'événement
        Event.RENDER
    } finally
    {
        removeEventListener ( Event.RENDER, miseAJour );
    }
}
}

```

La figure 11-7 illustre le résultat :



*Figure 11-7. Instance de la classe `Fenetre` affichée.*

Grâce à cette technique nous n'avons pas besoin d'écouter l'événement `Event.ADDED_TO_STAGE` afin de pouvoir cibler l'objet `Stage`. Mais comme nous l'avons vu précédemment, il est toujours recommandé d'écouter en interne l'événement `Event.ADDED_TO_STAGE` afin d'accéder à l'objet `Stage`. Cette technique permet donc aux objets non graphiques d'accéder à l'objet `Stage` de manière simplifiée.

En revanche, si nous devons réutiliser la classe `Fenetre` dans un autre projet, nous serons obligé de définir une classe du document nommée `Application` ainsi qu'une propriété statique `GLOBAL_STAGE` référençant l'objet `Stage`.

Afin d'automatiser ce processus, nous allons utiliser l'héritage.

### **Automatiser l'accès global à l'objet Stage**

Pour pouvoir bénéficier d'un accès global à l'objet `Stage` automatiquement dans tous nos projets, nous avons plusieurs possibilités.

La première, comme nous venons de voir à l'instant consiste à définir dans la classe du document de chaque projet, une propriété statique

accessible de n'importe classe. Permettant ainsi un accès simplifié à l'objet `Stage`. Il serait dommage de devoir recopier ce code dans classe du document, nous préférons donc utiliser la solution suivante.

Nous savons que la classe `Application` contient des fonctionnalités pouvant être nécessaires à chaque projet :

- Un accès global à l'objet `Stage`
- Une méthode `myGotoAndStop`

Afin d'hériter de ces fonctionnalités, nous allons utiliser pour chaque projet une classe du document héritant de la classe `Application`.

Il n'est pas nécessaire de dupliquer celle-ci dans le répertoire de classes de chaque projet nous allons l'isoler en la plaçant dans un répertoire spécifique global à tous nos projets.

A la racine de notre disque dur nous créons un répertoire nommé `classes_as3`. Puis nous créons répertoire `org` contenant un répertoire `abstrait`.

Au sein de ce dernier nous stockons la classe `ApplicationDefault` dont voici le code complet final :

```
package org.bytearray.abstrait
{
    import flash.display.MovieClip;
    import flash.events.Event;
    import flash.display.Stage;

    public class ApplicationDefault extends MovieClip
    {
        // point d'accès à l'objet Stage
        public static var GLOBAL_STAGE:Stage;
        // propriété permettant l'exécution de la fonction de rappel
        private var rappel:Function;

        public function ApplicationDefault ()
        {
            // affecte une référence à l'objet Stage
            ApplicationDefault.GLOBAL_STAGE = stage;
        }

        // méthode de déplacement de la tête de lecture personnalisé
        public function myGotoAndStop ( pImage:int, pFonction:Function ):void
        {
            // écoute de l'événement Event.RENDER
        }
    }
}
```



```
addEventListener ( Event.RENDER, miseAJour );

// déplacement de la tête de lecture
gotoAndStop ( pImage );

// retourne un objet permettant
rappel = pFonction;

// force la diffusion de l'événement Event.RENDER
stage.invalidate();

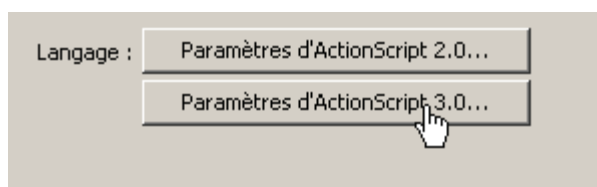
}

private function miseAJour ( pEvt:Event ):void
{
    // nous tentons d'appeler la fonction de rappel
    try
    {
        rappel();

        // si cela échoue, nous affichons un message d'erreur
    } catch ( pErreur:Error )
    {
        trace("Erreur : La méthode de rappel n'a pas été définie");

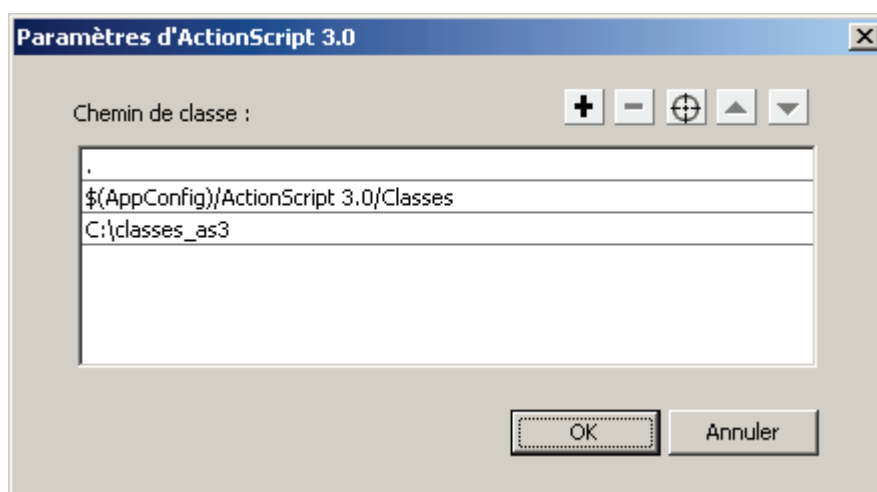
        // dans tout les cas, nous supprimons l'écoute de l'événement
        Event.RENDER
    } finally
    {
        removeEventListener ( Event.RENDER, miseAJour );
    }
}
}
```

Afin de spécifier un chemin d'accès de classes global à tous les projets au sein de Flash CS3, nous cliquons sur *Modifier* puis *Préférences* puis dans la liste nous sélectionnons *ActionScript*.



*Figure 11-8. Paramètres d'ActionScript 3.*

Une fois cliqué sur le bouton *Paramètres d'ActionScript 3* le panneau indiquant le chemin d'accès aux classes s'affiche. Comme l'illustre la figure 11-9.



*Figure 11-9. Chemin d'accès aux classes.*

En cliquant sur l'icône **+** nous ajoutons un champ afin de spécifier le chemin d'accès au répertoire contenant les classes globales. Dans notre exemple nous ajoutons une entrée contenant le chemin `C:\\classes_as3`.

Deux autres lignes sont déjà présentes, la première indique que le compilateur doit regarder à côté du document en cours. Puis la deuxième indique le chemin d'accès aux classes natives de Flash.

En cliquant sur OK, les classes stockées au sein du répertoire `classes_as3` seront disponibles depuis n'importe quel projet.

Nous allons modifier notre application en affectant une classe du document héritant de la classe `ApplicationDefault`.

Au sein du répertoire document nous créons une classe `Document` :

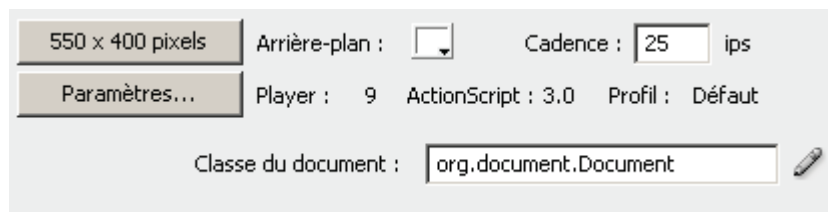
```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault
    {
        public function Document ()
        {
            trace( this );
        }
    }
}
```

```
}

```

Puis nous l'affectons comme classe du document en cours, comme l'indique la figure 11-10 :



*Figure 11-10. Affectation de la classe Document.*

A la compilation [object Document] s'affiche dans le panneau *Sortie*.

La classe `ApplicationDefault` devient ainsi une classe ne devant être qu'héritée. Une sorte de classe abstraite, même si comme nous l'avons vu lors du chapitre 8 intitulé *Programmation orientée objet*, ActionScript 3 ne permet pas la définition de vraie classe abstraite.

Tous les objets présents ou non au sein de la liste d'affichage devant faire référence à l'objet `Stage` devront cibler la propriété `ApplicationDefault.GLOBAL_STAGE`.

Nous modifions donc la classe `Fenetre` :

```
package org.bytearray.ui

{

    import flash.display.Shape;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Fenetre extends Shape

    {

        public function Fenetre ( pLargeur:Number, pHauteur:Number,
        pRayon:Number, pCouleur:Number )

        {

            graphics.beginFill ( pCouleur );
            graphics.drawRoundRect ( 0, 0, pLargeur, pHauteur, pRayon );

            // nous centrons la fenêtre
            // en utilisant la propriété statique Application.GLOBAL_STAGE
            // nous bénéficions d'un point d'accès global à l'objet Stage
            x = (ApplicationDefault.GLOBAL_STAGE.stageWidth - width)/2;
            y = (ApplicationDefault.GLOBAL_STAGE.stageHeight - height)/2;

        }

    }

}
```

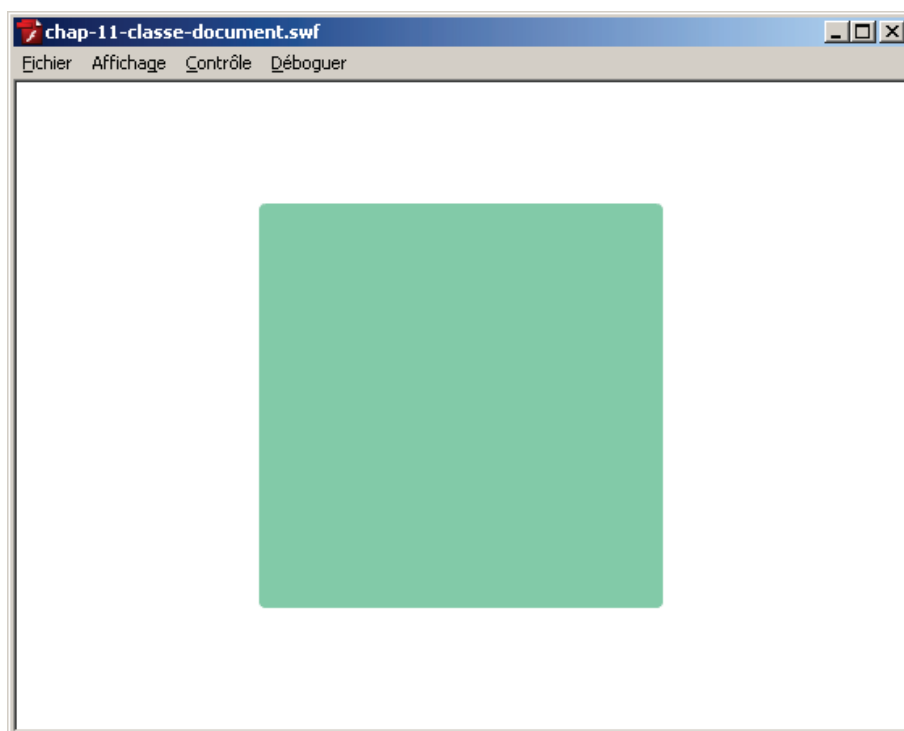
Puis nous modifions la classe `Document` afin d’instancier la classe `Fenetre` :

```
package org.bytearray.document
{
    import flash.events.Event;
    import org.bytearray.abstrait.ApplicationDefault;
    import org.bytearray.ui.Fenetre;

    public class Document extends ApplicationDefault
    {
        public function Document ()
        {
            // création de la fenêtre
            var maFenetre:Fenetre = new Fenetre( 250, 250, 8, 0x88CCAA );

            // ajout à la liste d'affichage
            addChild ( maFenetre );
        }
    }
}
```

La figure 11-11 illustre le résultat :



*Figure 11-11. Instance de la classe Fenetre.*

Souvenez-vous, la sous-classe n'hérite pas des propriétés statiques de la classe parente. Ainsi, la classe `Document` n'hérite pas de la propriété `GLOBAL_STAGE` de la classe `ApplicationDefault`.

### A retenir

- La fenêtre *Paramètres d'ActionScript 3* permet de définir un chemin d'accès global aux classes.

Nous allons nous intéresser au cours du prochain chapitre à la gestion des bitmap par programmation en ActionScript 3. Les classes `flash.display.Bitmap` et `flash.display.BitmapData` n'auront plus de secrets pour vous !

# 12

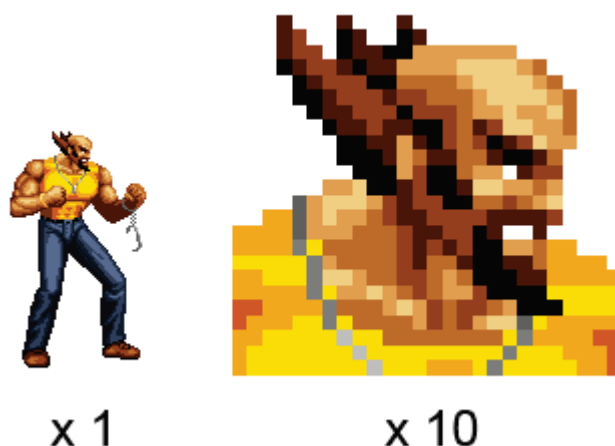
## Programmation Bitmap

<b>BITMAP ET VECTORIELS .....</b>	<b>2</b>
<b>COULEURS .....</b>	<b>3</b>
MANIPULER LES COULEURS .....	5
<b>LA CLASSE BITMAPDATA .....</b>	<b>7</b>
<b>CODAGE DE COULEURS.....</b>	<b>8</b>
<b>GERER LES RESSOURCES AVEC LE PROFILER .....</b>	<b>9</b>
<b>LA CLASSE BITMAP.....</b>	<b>12</b>
RÉUTILISER LES DONNÉES BITMAPS .....	15
<b>LIBERER LES RESSOURCES.....</b>	<b>18</b>
CALCULER LE POIDS D’UNE IMAGE EN MEMOIRE.....	26
LIMITATIONS MÉMOIRE .....	27
<b>IMAGES EN BIBLIOTHEQUE.....</b>	<b>28</b>
<b>PEINDRE DES PIXELS.....</b>	<b>30</b>
LIRE DES PIXELS.....	34
ACCROCHAGE AUX PIXELS .....	37
<b>LE LISSAGE .....</b>	<b>38</b>
<b>MISE EN CACHE DES BITMAP A L’EXECUTION .....</b>	<b>39</b>
EFFETS PERVERS .....	43
<b>FILTRE UN ÉLÉMENT VECTORIEL.....</b>	<b>46</b>
FILTRER UNE IMAGE BITMAP .....	59
ANIMER UN FILTRE.....	61
<b>RENDU BITMAP D’OBJET VECTORIELS .....</b>	<b>63</b>
<b>OPTIMISER LES PERFORMANCES.....</b>	<b>78</b>

## Bitmap et vectoriels

Avant d'entamer la découverte des fonctionnalités offertes par le lecteur Flash en matière de programmation bitmap, il convient de s'attarder sur le concept d'image bitmap et vectorielle.

Une image bitmap peut être considérée comme une grille constituée de pixels de couleur spécifique. En zoomant sur une image bitmap nous pouvons apercevoir chaque pixel la constituant.



*Figure 12-1. Image bitmap agrandie.*

A l'inverse, une image vectorielle n'est pas composée de pixels, mais de tracés issus de coordonnées mathématiques. Le lecteur Flash les interprète et dessine la forme correspondante.

La figure 12-2 illustre un exemple de tracé vectoriel :



*Figure 12-2. Image vectorielle agrandie.*

En cas d'agrandissement le tracé est recalculé, empêchant toute pixellisation de l'image quelque soit la résolution ou dimension. En matière de performances, l'affichage vectoriel requiert peu de mémoire mais peut nécessiter en cas de tracés complexes une forte sollicitation du processeur.

## Couleurs

L'espace de couleur utilisé dans Flash est l'ARVB, chaque couleur repose sur quatre composantes :

- L'alpha
- Le rouge
- Le vert
- Le bleu

Les trois composantes de couleurs donnent une combinaison de 16777215 couleurs possibles. L'espace colorimétrique RVB s'approche en réalité du nombre de couleurs maximum que l'œil de l'homme peut distinguer, ainsi le terme de *couleurs vraies* est couramment utilisé.



Chaque composante est codée sur 8 bits soit 1 octet et varie de 0 à 255. En binaire, une couleur ARVB peut être représentée de la manière suivante :

Alpha	Rouge	Vert	Bleu
11111111	11111111	00000000	00000000

Bien entendu, pour des questions de pratique nous travaillons généralement avec une base 16, plus couramment appelée représentation hexadécimale :

Alpha	Rouge	Vert	Bleu
FF	FF	00	00

Nous ajoutons le préfixe 0x devant une valeur hexadécimale afin de préciser au lecteur Flash qu'il s'agit d'une couleur encodée en base 16 :

```
// stocke une couleur hexadécimale
var couleur:Number = 0xFFFF0000;

// affiche : 4294901760
trace( couleur );
```

Pour générer une couleur aléatoire, nous pouvons évaluer un nombre aléatoire compris entre 0 et la couleur la plus haute, soit 0xFFFFFFFF :

```
// génère une couleur aléatoire
var couleurAleatoire:Number = Math.floor ( Math.random()*0xFFFFFFFF );

// affiche : 9019179
trace( couleurAleatoire );
```

Lorsque nous affichons la couleur évaluée, celle-ci est rendue par défaut sous une forme décimale. Si nous souhaitons obtenir une autre représentation de la couleur nous pouvons utiliser la méthode `toString` de la classe `Object`.

Celle-ci permet de convertir un nombre dans une base spécifique :

```
// génère une couleur aléatoire
var couleurAleatoire:Number = Math.floor ( Math.random()*0xFFFFFFFF );

// affiche la couleur au format hexadécimal (base 16)
// affiche : d419f6
trace( couleurAleatoire.toString(16) );

// affiche la couleur au format octal (base 8)
// affiche : 52267144
trace( couleurAleatoire.toString(8) );

// affiche la couleur au format binaire (base 2)
// affiche : 101010010110111001100100
trace( couleurAleatoire.toString(2) );
```

Une couleur RVB est représentée par une valeur hexadécimale à six chiffres. Deux chiffres étant nécessaires pour chaque composante :

```
var rouge:Number = 0xFF0000;
var vert:Number = 0x00FF00;
var bleu:Number = 0x0000FF;
```

Pour représenter une couleur ARVB, nous ajoutons le composant alpha en début de couleur :

```
var rougeTransparent:Number = 0x00FF0000;
var rougeSemiTransparent:Number = 0x88FF0000;
var rougeOpaque:Number = 0xFFFF0000;
```

Afin de faciliter la compréhension des couleurs dans Flash, nous allons nous attarder à présent sur leur manipulation.

## Manipuler les couleurs

La manipulation de couleurs est facilitée grâce aux opérateurs de manipulation binaire. Au cours de ce chapitre, nous allons travailler avec les différentes composantes de couleurs. Nous allons créer une classe `BitmapUtils` globale à tous nos projets, contenant différentes méthodes de manipulation.

Rappelez-vous, au cours du chapitre 11 intitulé *Classe du document*, nous avons créé un répertoire global de classes nommé `classes_as3`. Au sein du répertoire `org` du répertoire nous créons un répertoire nommé `utils`.

Puis nous définissons une classe `BitmapUtils` contenant une première méthode `hexArgb` :

```
package org.bytearray.utils
{
    import flash.display.BitmapData;

    public class BitmapUtils
    {
        public static function hexArgb ( pCouleur:Number ):Object
        {
            var composants:Object = new Object();
            // extraction de chaque composante
            composants.alpha = (pCouleur >>> 24) & 0xFF;
            composants.rouge = (pCouleur >>> 16) & 0xFF;
            composants.vert = (pCouleur >>> 8) & 0xFF;
            composants.bleu = pCouleur & 0xFF;

            return composants;
        }
    }
}
```

```
    }  
  }  
}
```

Grâce à la méthode `hexArgb`, l'extraction des composantes est simplifiée :

```
import org.bytearray.ouutils.BitmapOutils;  
  
// génère une couleur aléatoire  
var couleurAleatoire:Number = Math.floor ( Math.random()*0xFFFFFFFF );  
  
// affiche : 767D62D5  
trace( couleurAleatoire.toString(16).toUpperCase() );  
  
// extraction des composantes  
var composants:Object = BitmapOutils.hexArgb ( couleurAleatoire );  
  
var transparence:Number = composants.alpha;  
var rouge:Number = composants.rouge;  
var vert:Number = composants.vert;  
var bleu:Number = composants.bleu;  
  
// affiche : 76  
trace( transparence.toString(16).toUpperCase() );  
  
// affiche : 7D  
trace( rouge.toString(16).toUpperCase() );  
  
// affiche : 62  
trace( vert.toString(16).toUpperCase() );  
  
// affiche : D5  
trace( bleu.toString(16).toUpperCase() );
```

Nous pouvons aussi ajouter une méthode `argbHex` permettant d'assembler une couleur hexadécimale à partir de quatre composantes :

```
package org.bytearray.ouutils  
  
{  
    import flash.display.BitmapData;  
    public class BitmapOutils  
    {  
        public static function hexArgb ( pCouleur:Number ):Object  
        {  
            var composants:Object = new Object();  
            composants.alpha = (pCouleur >>> 24) & 0xFF;  
            composants.rouge = (pCouleur >>> 16) & 0xFF;  
            composants.vert  = (pCouleur >>> 8) & 0xFF;  
            composants.bleu  = pCouleur & 0xFF;  
        }  
    }  
}
```

```
        return composants;
    }

    public static function argbHex ( pAlpha:int, pRouge:int, pVert:int,
    pBleu:int ):uint
    {
        return pAlpha << 24 | pRouge << 16 | pVert << 8 | pBleu;
    }
}
}
```

Une fois la méthode `argbHex` définie, nous pouvons générer une couleur aléatoire à partir de quatre composantes :

```
import org.bytearray.ouils.BitmapOutils;

var transparence:Number = Math.floor ( Math.random()*256 );
var rouge:Number = Math.floor ( Math.random()*256 );
var vert:Number = Math.floor ( Math.random()*256 );
var bleu:Number = Math.floor ( Math.random()*256 );

// assemble la couleur
var couleur:Number = BitmapOutils.ArgbHex ( transparence, rouge, vert, bleu
);

// affiche : 3F31D4B2
trace( couleur.toString(16).toUpperCase() );
```

Notre classe `BitmapOutils` sera très vite enrichie, nous y ajouterons bientôt de nouvelles fonctionnalités. Libre à vous d'ajouter par la suite différentes méthodes facilitant la manipulation des couleurs.

## A retenir

- Une couleur est constituée de 4 composants codés sur 8 bits, soit un octet.
- Chaque composant varie entre 0 et 255.
- Pour changer la base d'un nombre, nous utilisons la méthode `toString` de la classe `Object`.
- La manipulation des couleurs est facilitée grâce aux opérateurs de manipulation binaire.

## La classe `BitmapData`

La classe `BitmapData` fut introduite avec le lecteur Flash 8 et permet la création d'images bitmaps par programmation. En ActionScript 3, son utilisation est étendue car toutes les données bitmaps sont représentées par la classe `flash.display.BitmapData`.

L'utilisation d'images bitmaps par programmation permet de travailler sur les pixels et de créer toutes sortes d'effets complexes qui ne peuvent être réalisés à l'aide de vecteurs.

Afin de créer dynamiquement une image bitmap, nous devons tout d'abord créer les pixels la composant. Pour cela nous utilisons la classe `flash.display.BitmapData` dont voici la signature du constructeur :

```
public fonction BitmapData(width:int, height:int, transparent:Boolean = true,
fillColor:uint = 0xFFFFFFFF)
```

Celui-ci accepte quatre paramètres :

- `width` : largeur de l'image.
- `height` : hauteur de l'image.
- `transparent` : un booléen indiquant la transparence de l'image. Transparent par défaut
- `fillColor` : la couleur du bitmap en 32 ou 24 bits. Couleur blanche par défaut.

Pour créer une image transparente de 1000 \* 1000 pixels de couleur beige nous écrivons le code suivant :

```
// création d'une image de 1000 * 1000 pixels, transparente de couleur beige
var monImage:BitmapData = new BitmapData (1000, 1000, true, 0x00F0D062);
```

Aussitôt l'image créée, les données bitmaps sont stockées en mémoire.

## Codage de couleurs

Lorsqu'une image est créée, le lecteur Flash stocke la couleur de chaque pixel en mémoire. Chacun d'entre eux est codé sur 32 bits, 4 octets sont donc nécessaires à la description d'un pixel.

Afin d'illustrer ce comportement, nous créons une première image bitmap semi transparente de 1000 \* 1000 pixels :

```
// création d'une image transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, true, 0xAAF0D062 );

// récupère la couleur d'un pixel
var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );

// extraction du canal alpha
var transparence:Number = BitmapOutils.hexArgb ( couleurPixel ).alpha;

// affiche : 170
trace( transparence );
```

En isolant le canal alpha, nous voyons que son intensité vaut 170. Dans un souci d'optimisation nous pourrions décider de créer une image non transparente, pensant que celle-ci serait codée sur 24 bits :

```
// création d'une image non transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, false, 0xF0D062 );

// récupère la couleur d'un pixel
var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );

// extraction du canal alpha
var transparence:Number = BitmapUtils.hexArgb ( couleurPixel ).alpha;

// affiche : 255
trace( transparence );
```

Dans le cas d'une image non transparente, l'intensité du canal alpha est automatiquement définie à 255. Si nous tentons de passer une couleur 32 bits, les 8 premiers bits sont ignorés :

```
// création d'une image non transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, false, 0xBBF0D062 );

// récupère la couleur d'un pixel
var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );

// récupère le canal alpha
var transparence:Number = BitmapUtils.hexArgb ( couleurPixel ).alpha;

// affiche : 255
trace( transparence );
```

Au sein du lecteur Flash, quelque soit la transparence de l'image, les couleurs sont toujours codées sur 32 bits. La création d'une image opaque n'entraîne donc aucune optimisation mémoire mais facilite en revanche l'affichage.

## A retenir

- Pour créer une image par programmation nous utilisons la classe `BitmapData`.
- Chaque couleur de pixel composant une instance de `BitmapData` est codée sur 32 bits.
- 1 pixel pèse 4 octets en mémoire.
- La création d'image opaque n'optimise pas la mémoire mais facilite le rendu de l'image.

## Gérer les ressources avec le profiler

Afin d'optimiser un projet ActionScript 3, il est impératif de maîtriser la gestion des ressources. L'utilisation de la classe `BitmapData` nécessite une attention toute particulière quant à l'occupation mémoire engendrée par son utilisation.

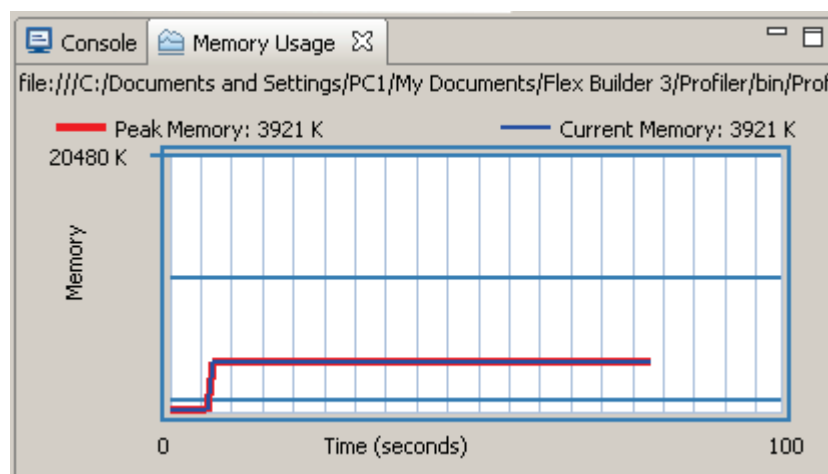
Pour tester le comportement du lecteur Flash nous utiliserons le profiler de Flex Builder 3, qui est aujourd'hui l'outil le plus adapté en matière de gestion et optimisation des ressources.

Le profiler de Flex Builder 3 est un module permettant de connaître en temps réel l'occupation mémoire de chaque objet ainsi que l'occupation mémoire totale d'une application ActionScript 3. Il n'existe malheureusement pas d'outil similaire dans Flash CS3. Lorsque vous souhaitez tester les ressources d'un projet ActionScript 3 développé avec Flash CS3, vous avez la possibilité de charger celle-ci au sein d'une application Flex, afin de bénéficier du profiler.

Nous allons analyser la mémoire utilisée par le lecteur Flash en créant une première image bitmap transparente :

```
// création d'une image transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, true, 0x00F0D062 );
```

La figure 12-4 illustre la fenêtre *Utilisation Mémoire* du profiler :



*Figure 12-4. Création de données bitmaps transparente.*

Nous voyons la courbe augmenter sensiblement lors de la création de l'instance de `BitmapData`. Celle-ci occupe environ 3906 Ko en mémoire vive.

Afin d'analyser ce résultat, faisons un tour d'horizon des différentes légendes du panneau *Utilisation mémoire* :

- **Peak Memory** (Seuil maximum atteint) : plus haute occupation mémoire atteinte depuis le lancement de l'animation.
- **Current Memory** (Mémoire actuelle) : occupation mémoire courante.
- **Time** (Temps) : nombre de secondes écoulées depuis le lancement de l'animation.

Le profiler nous indique que l'image créée, occupe en mémoire environ 3,9 Mo. Nous pouvons facilement vérifier cette valeur avec le calcul suivant :

```
1000 * 1000 = 1000000 pixels
```

Chaque pixel est codé sur 32 bits (4 octets) :

```
1000000 pixels * 4 octets = 4000000 octets
```

Afin d'obtenir l'équivalent en Ko, nous divisons par 1024 :

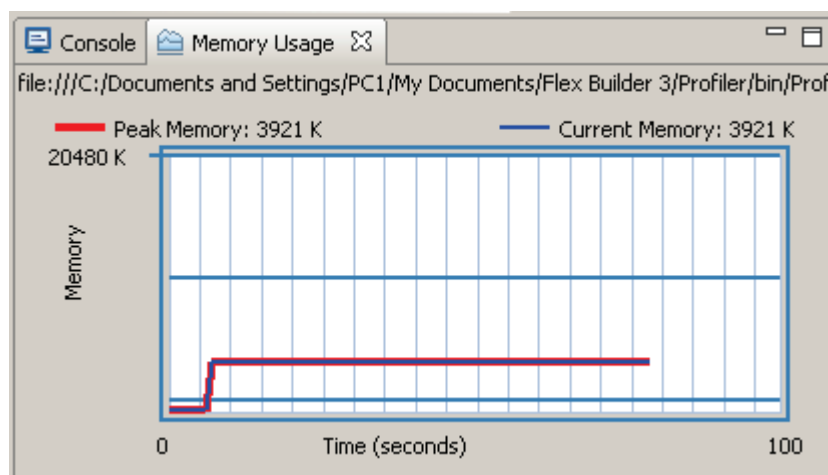
```
4000000 / 1024 = 3906,25 Ko
```

Nous retrouvons le poids indiqué par le profiler. Nous verrons très bientôt comment faciliter ce calcul au sein d'une application ActionScript.

Comme nous l'avons vu précédemment, le lecteur Flash code, quelque soit la transparence de l'image, les couleurs sur 32 bits. Si nous créons une image non transparente, l'occupation mémoire reste la même :

```
// création d'une image non transparente
var monImage:BitmapData = new BitmapData ( 1000, 1000, false, 0xF0D062 );
```

La figure 12-4 illustre le résultat :



*Figure 12-4. Création de données bitmaps non transparentes.*

Le profiler est un outil essentiel au déboguage d'applications ActionScript 3. Certains outils tiers existent mais n'offrent pas une telle granularité dans les informations apportées.

## A retenir



- Le Profiler est un outil intégré à Flex Builder 3 facilitant la gestion des ressources d'un projet ActionScript 3.
- Il n'existe pas d'outil intégré équivalent dans Flash CS3.

## La classe Bitmap

Comme son nom l'indique, la classe `BitmapData` représente les données bitmaps mais celle-ci ne peut être affichée directement. Afin de rendre une image nous devons associer l'instance de `BitmapData` à la classe `flash.display.Bitmap`.

Il est important de considérer la classe `Bitmap` comme simple conteneur, celle-ci sert à *présenter* les données bitmaps.

Dans les précédentes versions d'ActionScript la classe `MovieClip` était utilisée pour afficher l'image :

```
// création d'un clip conteneur
var monClipConteneur:MovieClip = this.createEmptyMovieClip ("conteneur", 0);

// création des données bitmaps
var donneesBitmap:BitmapData = new BitmapData (300, 300, false, 0xFF00FF);

// affichage de l'image
monClipConteneur.attachBitmap (donneesBitmap, 0);
```

En ActionScript 3, nous utilisons la classe `Bitmap` dont voici le constructeur :

```
Bitmap(bitmapData:BitmapData = null, pixelSnapping:String = "auto",
smoothing:Boolean = false)
```

Celui-ci accepte trois paramètres :

- `bitmapData` : les données bitmaps à afficher.
- `pixelSnapping` : accrochage aux pixels.
- `Smoothing` : un booléen indiquant si l'image doit être lissée ou non.

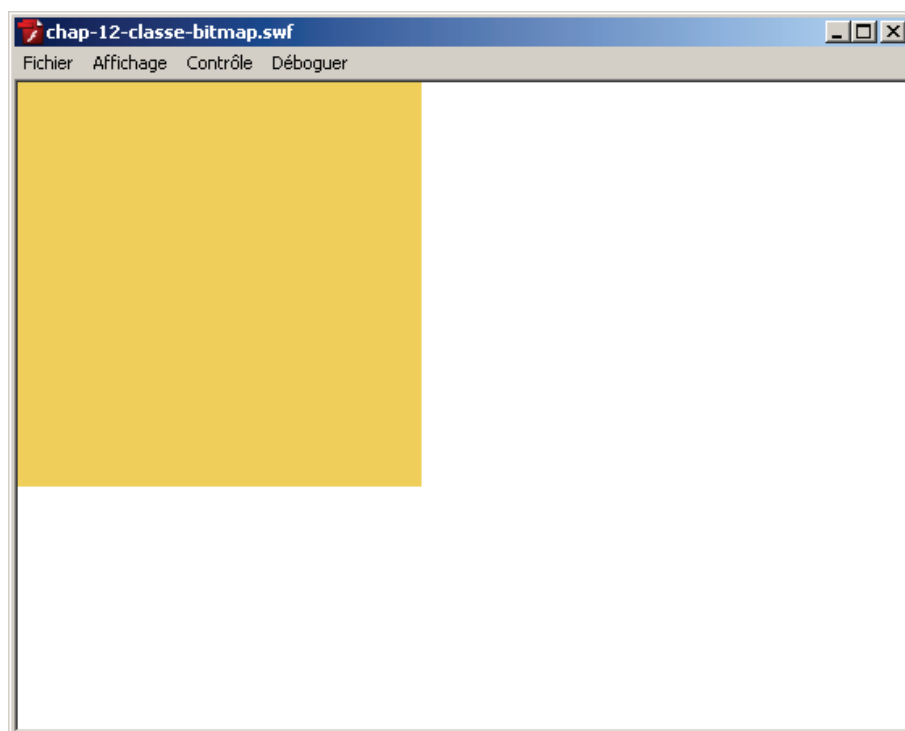
Nous passons en premier paramètre l'instance de `BitmapData` à afficher, les deux autres paramètres seront traités plus loin :

```
// création d'une image de 250 * 250 pixels, non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// création d'un conteneur pour l'image bitmap
var monConteneurImage:Bitmap = new Bitmap ( monImage );

// ajout du conteneur
addChild ( monConteneurImage );
```

En testant le code précédent, nous obtenons le résultat illustré en figure 12-5 :



*Figure 12-5. Affichage d'une instance de  
BitmapData.*

Si nous souhaitons modifier la présentation des données bitmaps, nous utilisons les différentes propriétés de la classe `Bitmap`. À l'inverse, si nous devons travailler sur les pixels composant l'image, nous utiliserons les méthodes de la classe `BitmapData`.

De par l'héritage, toutes les propriétés de la classe `DisplayObject` sont donc disponibles sur la classe `Bitmap` :

```
// création d'une image de 250 * 250 pixels, non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// création d'un conteneur pour l'image bitmap
var monConteneurImage:Bitmap = new Bitmap ( monImage );

// ajout du conteneur
addChild ( monConteneurImage );

// positionnement et redimensionnement
monConteneurImage.x = 270;
monConteneurImage.y = 120;
monConteneurImage.scaleX = .5;
monConteneurImage.scaleY = .5;
monConteneurImage.rotation = 45;
```

Le code précédent déplace l'image, réduit l'image et lui fait subir une rotation de 45 degrés. Le résultat est illustré en figure 12-6 :

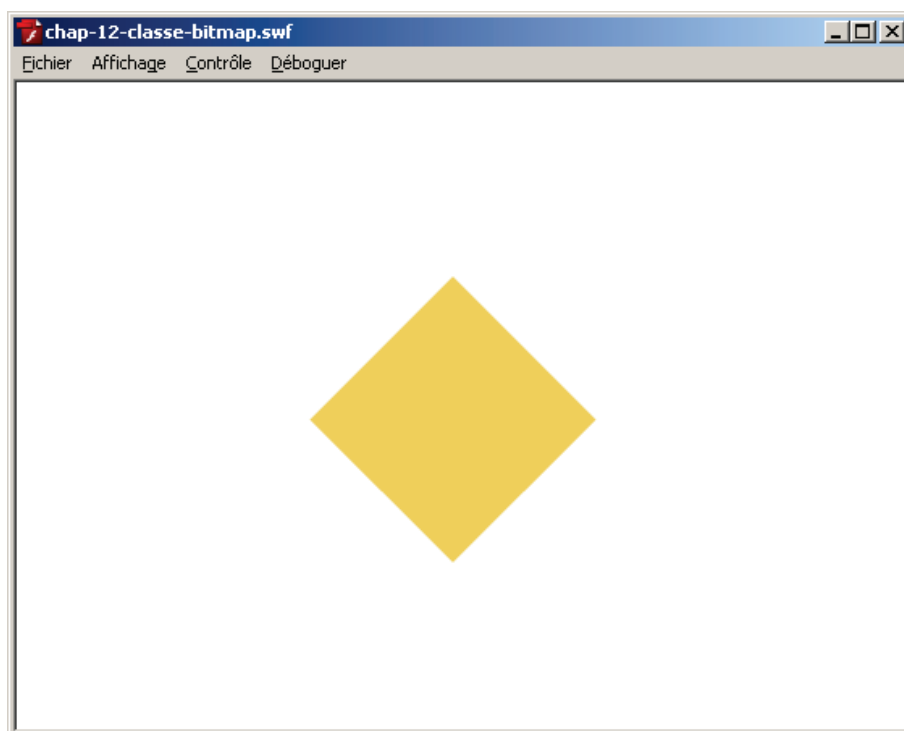


Figure 12-6. Déplacement et rotation d'une image bitmap.

Il était impossible d'accéder aux données bitmaps associées à un `MovieClip` dans les précédentes versions d'ActionScript. En ActionScript 3, nous utilisons la propriété `bitmapData` de la classe `Bitmap` :

```
// création d'une image de 250 * 250 pixels, non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// création d'un conteneur pour l'image bitmap
var monConteneurImage:Bitmap = new Bitmap ( monImage );

// ajout du conteneur
addChild ( monConteneurImage );

// positionnement et redimensionnement
monConteneurImage.x = 270;
monConteneurImage.y = 120;
monConteneurImage.scaleX = .5;
monConteneurImage.scaleY = .5;
monConteneurImage.rotation = 45;

var donneesBitmap:BitmapData = monConteneurImage.bitmapData;

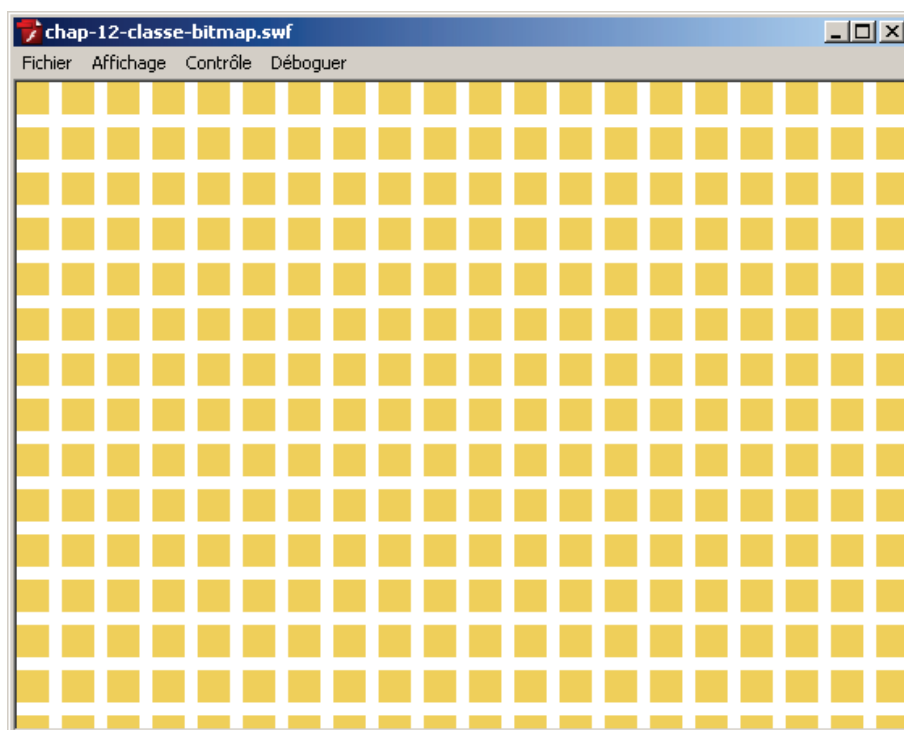
// affiche : 250
trace(donneesBitmap.width );
// affiche : 250
trace(donneesBitmap.height );
```

Nous remarquons que les dimensions de l'instance de `BitmapData` ne sont pas altérées par le redimensionnement de l'objet `Bitmap`.

Souvenez-vous, la classe `Bitmap` *présente* simplement les données bitmaps définies par la classe `BitmapData`.

## Réutiliser les données bitmaps

Dans certaines applications, les données bitmaps peuvent être réutilisées lorsque les pixels sont identiques mais présentés différemment. Imaginons que nous souhaitons construire un damier comme celui illustré en figure 12-7 :



*Figure 12-7. Damier constitué d'instances de `BitmapData`.*

Nous pourrions être tentés d'écrire le code suivant :

```
for ( var i:int = 0; i< 300; i++ )
{
    // création d'un carré de 20 * 20 pixels, non transparent de couleur beige
    var monImage:BitmapData = new BitmapData (20, 20, false, 0xF0D062);

    // création d'un conteneur pour l'image bitmap
    var monConteneurImage:Bitmap = new Bitmap ( monImage );

    // ajout du conteneur à la liste d'affichage
    addChild ( monConteneurImage );

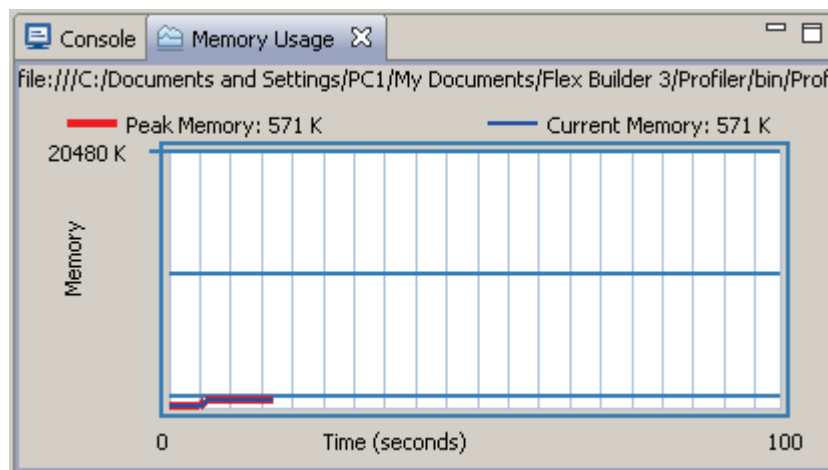
    // positionnement des conteneurs d'images
    monConteneurImage.x = ( monConteneurImage.width + 8 ) * Math.round (i %
20);
```

```

        monConteneurImage.y = ( monConteneurImage.height + 8 ) * Math.floor (i /
20);
    }

```

Pour chaque itération nous créons une instance de `BitmapData`, puis un objet `Bitmap` afin d'afficher chaque image. Si nous regardons l'occupation mémoire. Lorsque le damier est créé, l'occupation mémoire est de 571 Ko.



*Figure 12-8. Occupation mémoire sans optimisation du damier.*

Le code précédent n'est pas optimisé car nous créons à chaque itération une instance de `BitmapData` de 20 pixels pesant 1,56 Ko. En instanciant 300 fois ces données bitmaps, nous obtenons un poids total cumulé pour les images d'environ 470 Ko.

Souvenez-vous, la classe `Bitmap` permet d'afficher des données bitmaps, il est donc tout à fait possible d'afficher plusieurs images à partir d'une même instance de `BitmapData`.

Nous pourrions obtenir le même damier en divisant le poids de presque 6 fois :

```

// création d'une seule instance de BitmapData en dehors de la boucle
var monImage:BitmapData = new BitmapData (20, 20, false, 0xF0D062);

for ( var i:int = 0; i< 300; i++ )
{
    // création d'un conteneur pour les données bitmaps
    var monConteneurImage:Bitmap = new Bitmap ( monImage );

    // ajout du conteneur
    addChild ( monConteneurImage );

    // positionnement des conteneurs de bitmap
}

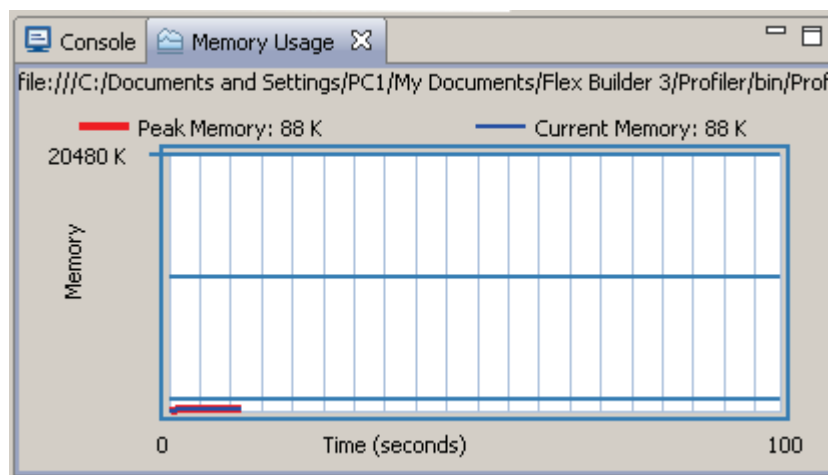
```

```

        monConteneurImage.x = ( monConteneurImage.width + 8 ) * Math.round ( i %
20);
        monConteneurImage.y = ( monConteneurImage.height + 8 ) * Math.floor ( i /
20);
    }

```

Avec le code précédent l'occupation mémoire a chuté, nous passons à environ 88 Ko d'occupation mémoire.



*Figure 12-9. Occupation mémoire avec optimisation du damier.*

Il est donc fortement recommandé de réutiliser les données bitmaps lorsque nous souhaitons afficher plusieurs fois les mêmes données bitmaps, même sous une forme différente.

Nous pourrions modifier la *présentation* des données bitmaps grâce aux différentes propriétés de la classe `Bitmap`. Dans le code suivant nous modifier la taille et la rotation de chaque élément du damier :

```

// création d'une seule instance de BitmapData en dehors de la boucle
var monImage:BitmapData = new BitmapData (20, 20, false, 0xF0D062);

for ( var i:int = 0; i< 300; i++ )
{
    // création d'un conteneur pour les données bitmaps
    var monConteneurImage:Bitmap = new Bitmap ( monImage );

    // ajout du conteneur
    addChild ( monConteneurImage );

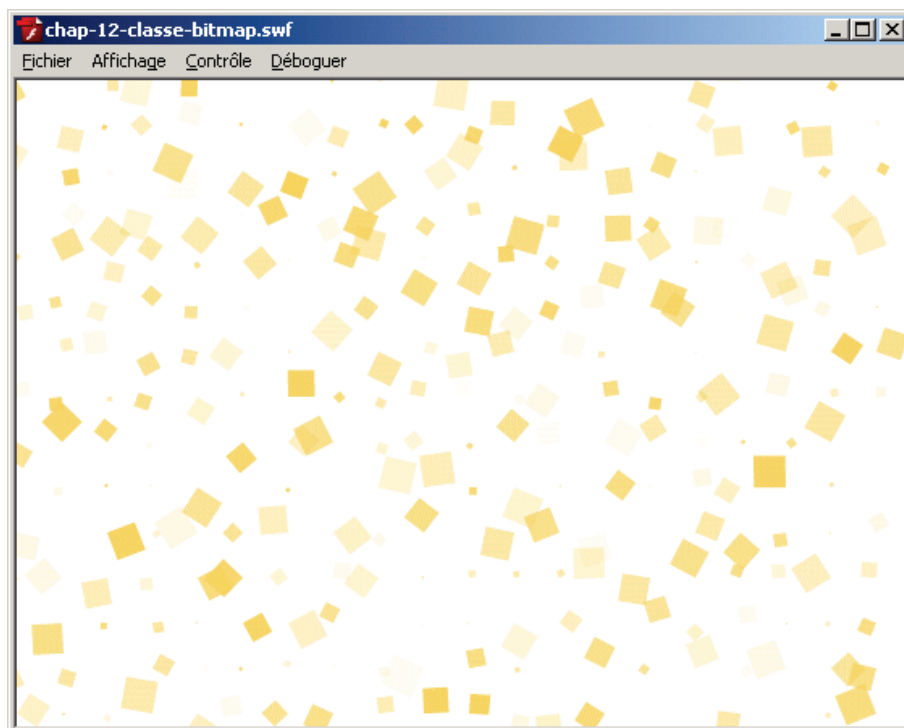
    // positionnement des conteneurs de bitmap
    monConteneurImage.x = ( monConteneurImage.width + 8 ) * Math.round ( i %
20);
    monConteneurImage.y = ( monConteneurImage.height + 8 ) * Math.floor ( i /
20);

    // taille, rotation et transparence aléatoires

```

```
monConteneurImage.scaleX = monConteneurImage.scaleY = Math.random();  
monConteneurImage.rotation = Math.floor ( Math.random()*360 );  
monConteneurImage.alpha = Math.random();  
}
```

La figure 12-10 illustre le résultat :



*Figure 12-10. Damier constitué d'une seule instance de  
BitmapData.*

Ce décor est constitué des mêmes données bitmaps, mais présentées sous différentes formes.

## A retenir

- La classe `Bitmap` permet de présenter les données bitmaps définies par la classe `BitmapData`.
- Il est important de réutiliser les données bitmaps lorsque cela est possible.

## Libérer les ressources

Comme nous l'avons vu lors du chapitre 2 intitulé *Langage et API du lecteur Flash* lorsqu'un objet n'est pas référencé, celui-ci est aussitôt considéré comme éligible à la suppression par le *ramasse-miettes*.

Dans le code suivant nous créons une instance de `BitmapData` à partir de 5 secondes, aucune référence n'est conservée :

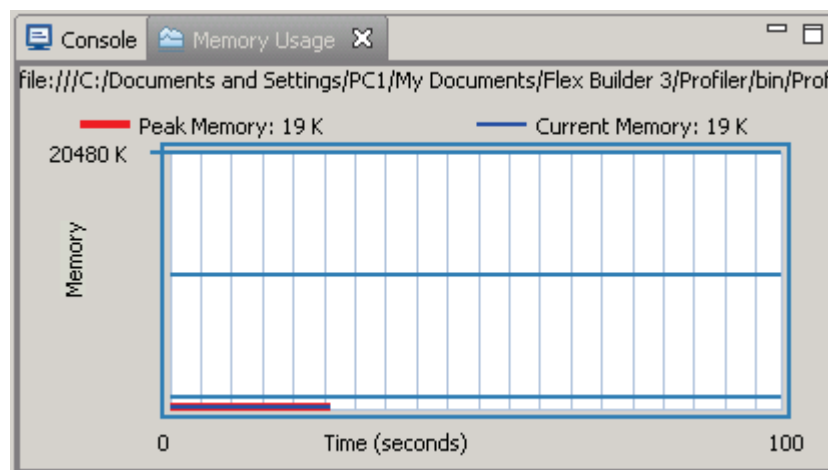
```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, creation );

// démarrage du minuteur
minuteur.start();

function creation ( pEvt:TimerEvent ):void
{
    // création d'une image non transparente de 350 * 350 pixels
    var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );
}
```

Une fois la fonction `creation` exécutée, la variable locale `monBitmap` expire, les données bitmaps sont aussitôt supprimées de la mémoire. En testant le code précédent nous ne remarquons aucune augmentation de l'occupation mémoire :



*Figure 12-11. Occupation mémoire de l'application.*

Si nous ajoutons à la liste d'affichage chaque instance de `BitmapData` créée, les données bitmaps sont alors référencées et ne sont plus éligibles à la suppression :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, creation );

// démarrage du minuteur
minuteur.start();
```

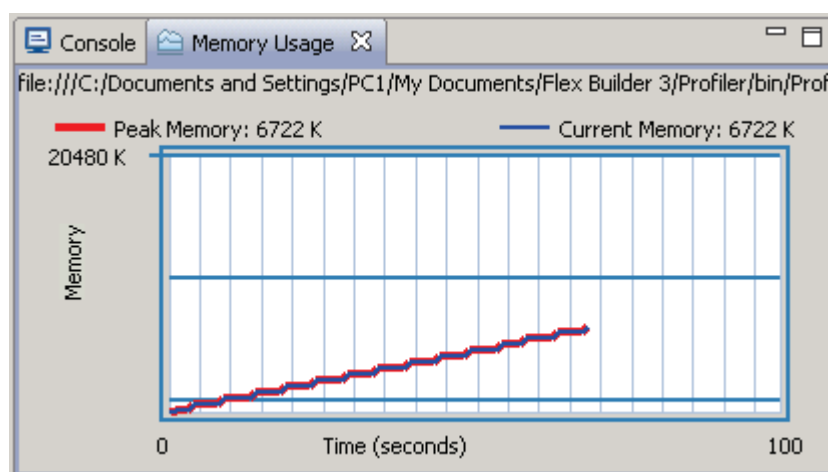


```
function creation ( pEvt:TimerEvent ):void
{
    // création d'une image non transparente de 350 * 350 pixels
    var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );

    // création de l'enveloppe Bitmap
    var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

    // ajout à la liste d'affichage
    addChild ( monConteneurBitmap );
}
```

La figure 12-12 montre l'augmentation de l'occupation mémoire :



*Figure 12-12. Augmentation de l'occupation mémoire de l'application.*

Toutes les 5 secondes, une image bitmap est créée puis ajoutée à la liste d'affichage. A partir d'une minute et dix secondes nous obtenons une occupation mémoire d'environ 6,7 Mo. Chaque instance nécessitant environ 478,5 Ko.

Lorsque nous n'avons plus besoin d'une image, il est fortement recommandé de la désactiver afin de libérer la mémoire. Pour cela, nous disposons de plusieurs solutions.

La première, consiste à désactiver l'image et libérer la mémoire en utilisant la méthode `dispose` de la classe `BitmapData`.

Dans le code suivant, un minuteur crée des instances de `BitmapData` toutes les 5 secondes puis les ajoutent à la liste d'affichage. A partir de 30 secondes nous stoppons la création des images et appelons la méthode `dispose` sur chaque instance de `BitmapData` :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );
```

```
// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, creation );

// démarrage du minuteur
minuteur.start();

function creation ( pEvt:TimerEvent ):void
{

    // création d'une image non transparente de 350 * 350 pixels
    var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );

    // création de l'enveloppe Bitmap
    var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

    // ajout à la liste d'affichage
    addChild ( monConteneurBitmap );

}

var minuteurNettoyage:Timer = new Timer ( 30000, 1 );

minuteurNettoyage.addEventListener( TimerEvent.TIMER, nettoyage );

minuteurNettoyage.start();

// désactivation des données bitmaps à partir de 30 secondes
function nettoyage ( pEvt:TimerEvent ):void
{
    minuteur.stop();

    var lng:int = numChildren;

    var monImage:Bitmap;

    while ( lng-- )
    {
        monImage = Bitmap ( getChildAt ( lng ) );

        // désactive les données bitmaps
        monImage.bitmapData.dispose();
    }
}
```

En analysant les informations fournies par le profiler, nous voyons que la mémoire n'est pas libérée :

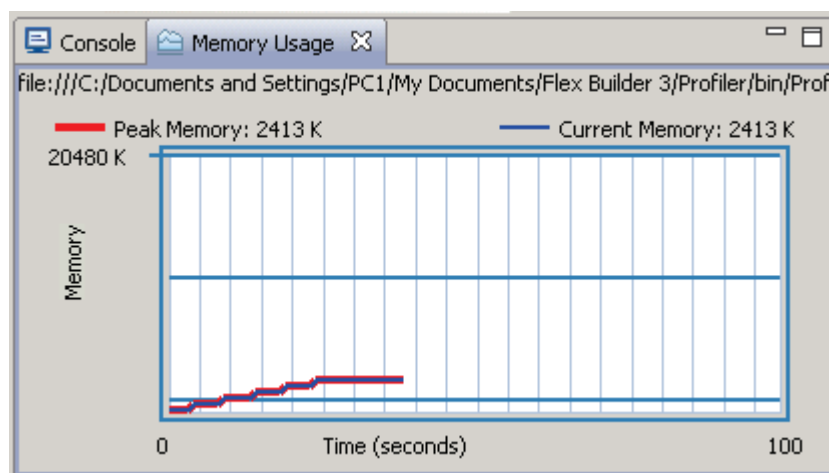


Figure 12-13. Occupation mémoire de l'application.

Il s'agit en réalité d'un bogue du profiler, qui n'affiche pas correctement la libération de la mémoire lors de l'utilisation de la méthode `dispose`. A l'aide d'autres outils, nous remarquons que la mémoire est libérée immédiatement.

Attention, bien que l'appel à la méthode `dispose` supprime visuellement les images et libère la mémoire, celles-ci sont toujours présentes au sein de la liste d'affichage et donc référencées. Afin de totalement désactiver une instance de `BitmapData` il faut veiller à supprimer l'instance de la liste d'affichage, puis appeler la méthode `dispose`.

Nous modifions le code précédent en supprimant chaque instance de l'affichage puis en appelant la méthode `dispose` :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 0 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, execution );

// démarrage du minuteur
minuteur.start();

// création et ajout à l'affichage d'une instance de BitmapData
// toutes les 5 secondes
function execution ( pEvt:TimerEvent ):void
{
    // création d'une image non transparente de 350 * 350 pixels
    var monBitmap:BitmapData = new BitmapData ( 350, 350, false, 0x990000 );

    // création de l'enveloppe Bitmap
    var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

    // ajout à la liste d'affichage
    addChild ( monConteneurBitmap );
}
```

```
}

var minuteurNettoyage:Timer = new Timer ( 30000, 1 );

minuteurNettoyage.addEventListener( TimerEvent.TIMER, nettoyage );

minuteurNettoyage.start();

// libération des ressources au bout de 30 secondes
function nettoyage ( pEvt:TimerEvent ):void
{
    minuteur.stop();

    var lng:int = numChildren;

    var monImage:Bitmap;

    while ( lng-- )
    {
        monImage = Bitmap ( removeChildAt ( lng ) );

        // désactive les données bitmaps
        monImage.bitmapData.dispose();
    }
}
```

L'approche suivante s'appuie sur le ramasse-miettes en supprimant les références pointant vers les instances de `BitmapData`. Cette technique a pour inconvénient de ne pas supprimer immédiatement les données bitmaps en mémoire. Elles le seront *uniquement* si le ramasse-miettes procède à un nettoyage. Souvenez-vous que celui-ci peut ne **jamais** intervenir.

Dans certaines applications, les données bitmaps peuvent être utilisées sans être affichées. Dans le cas d'une application d'encodage et de compression d'images, un tableau de références est généralement créé afin d'accéder rapidement à chaque instance :

```
// création d'un minuteur
var minuteur:Timer = new Timer ( 5000, 5 );

// écoute de l'événement TimerEvent.TIMER
minuteur.addEventListener( TimerEvent.TIMER, execution );

// démarrage du minuteur
minuteur.start();

// conteneur de références
var tableauImages:Array = new Array();

// création d'une instance de BitmapData
// toutes les 5 secondes
```

```
function execution ( pEvt:TimerEvent ):void
{
    // création d'une image non transparente de 350 * 350 pixels
    var monBitmap:BitmapData = new BitmapData ( 350, 350, false,
    Math.random()*0xFFFFFFFF );

    // création de l'enveloppe Bitmap
    var monConteneurBitmap:Bitmap = new Bitmap ( monBitmap );

    // stockage des références
    tableauImages.push ( monConteneurBitmap );
}

var minuteurNettoyage:Timer = new Timer ( 30000, 1 );
minuteurNettoyage.addEventListener( TimerEvent.TIMER, nettoyage );
minuteurNettoyage.start();

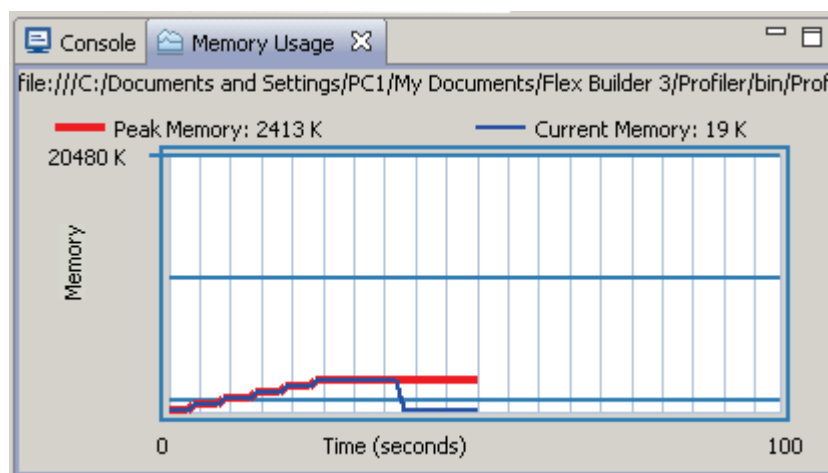
// libération des ressources au bout de 30 secondes
function nettoyage ( pEvt:TimerEvent ):void
{
    minuteur.stop();

    var lng:int = tableauImages.length;

    while ( lng-- )
    {
        // supprime chaque référence
        tableauImages [ lng ] = null;
    }
}
```

Les seules références aux instances de `BitmapData` ne résident pas au sein de la liste d’affichage mais au sein du tableau `tableauImages`. Pour libérer les ressources, nous passons chaque référence à `null`.

Lorsque le *ramasse-miettes* intervient, les ressources sont libérées :



*Figure 12-14. Chute de l'occupation mémoire de l'application.*

En utilisant cette technique il n'est pas nécessaire d'appeler la méthode `dispose`.

Afin d'optimiser nos tests nous pouvons grâce au profiler déclencher le *ramasse-miettes* et voir si les ressources sont bien libérées lors de son passage. Il n'est pas possible officiellement de déclencher le *ramasse-miettes* par programmation.

Une fois une image désactivée, celle-ci ne peut plus être utilisée. Dans le code suivant, nous tentons de cloner une image bitmap désactivée :

```
// création d'une image de 250 * 250 pixels
// non transparente de couleur beige
var monImage:BitmapData = new BitmapData (250, 250, false, 0xF0D062);

// désactivation des données bitmaps
monImage.dispose();

// tentative de clonage des données bitmaps
// affiche : ArgumentError: Error #2015: BitmapData non valide.
monImage.clone();
```

L'appel de la méthode `clone` lève une erreur de type `ArgumentError`. Une fois désactivée, il est impossible de réactiver l'image.

## A retenir

- La méthode `dispose` permet de désactiver une image et de libérer instantanément la mémoire utilisée. Cette technique est recommandée.
- En supprimant simplement les références pointant vers l'image nous ne sommes pas garantis que la mémoire soit libérée. Nous sommes tributaires du ramasse-miettes.

## Calculer le poids d'une image en mémoire

Afin de faciliter la manipulation de données bitmaps par programmation nous allons ajouter une méthode nommée `poids` au sein de la classe `BitmapUtils` :

```
package org.bytearray.outils

{
    import flash.display.BitmapData;

    public class BitmapUtils
    {
        public static function hexArgb ( pCouleur:Number ):Object
        {
            var composants:Object = new Object();
            composants.alpha = (pCouleur >>> 24) & 0xFF;
            composants.rouge = (pCouleur >>> 16) & 0xFF;
            composants.vert = (pCouleur >>> 8) & 0xFF;
            composants.bleu = pCouleur & 0xFF;

            return composants;
        }

        public static function argbHex ( pAlpha:int, pRouge:int, pVert:int,
pBleu:int ):uint
        {
            return ( pAlpha << 24 | pRouge << 16 | pVert << 8 | pBleu );
        }

        public static function poids ( pBitmapData:BitmapData ):Number
        {
            return (pBitmapData.width * pBitmapData.height) * 4;
        }
    }
}
```

La méthode `poids` nous renvoie le poids de l'image en mémoire en octets :

```
import org.bytearray.ouils.BitmapOutils;

// création d'une instance de BitmapData
var monBitmap:BitmapData = new BitmapData ( 1000, 1000, false,
Math.random()*0xFFFFFFFF );

// calcul du poids en mémoire
var poids:Number = BitmapOutils.poids ( monBitmap ) / 1024;

// affiche : 3906.25
trace( poids );
```

La classe `BitmapOutils` pourra ainsi être réutilisée à tout moment dans différents projets.

## Limitations mémoire

Pour des raisons de performances, la taille maximale d'une instance de `BitmapData` créée par programmation est limitée à 2880 \* 2880 pixels. Une image d'une telle dimension nécessite près de 32 Mo de mémoire vive, ce qui représente une occupation mémoire non négligeable :

```
import org.bytearray.ouils.BitmapOutils;

// création d'une image de 2880 * 2880 pixels
// non transparente de couleur beige
var monImage:BitmapData = new BitmapData (2880, 2880, false, 0xF0D062);

var poidsImage:Number = BitmapOutils.poids ( monImage );

// affiche : 32400
trace( poidsImage / 1024 );
```

Si nous tentons tout de même de créer une image d'une taille supérieure, le lecteur Flash lève une erreur à l'exécution :

```
// lève l'erreur à l'exécution suivante :
// Error #2015: BitmapData non valide.
var monImage:BitmapData = new BitmapData (3000, 3000, false, 0xF0D062);
```

Dans le cas d'une application de dessin, nous pourrions indiquer à l'utilisateur que l'image créée est trop grande en gérant l'exception :

```
try
{
    var monImage:BitmapData = new BitmapData (3000, 3000, false, 0xF0D062);
} catch ( pError:Error )
{
    // affiche : ArgumentError: Error #2015: BitmapData non valide.
    trace( pError );
}
```



```
    trace("Image trop grande !");  
}
```

Dans le cas d'une application nécessitant des images de dimensions supérieures, plusieurs instances de `BitmapData` peuvent être utilisées afin de contourner cette limitation.

### A retenir

- La taille maximale d'une instance de `BitmapData` créée par programmation est de 2880\*2880 pixels.

## Images en bibliothèque

Lorsqu'une image est présente au sein de la bibliothèque. Celle-ci est assimilée à une instance de `BitmapData`. Dans un nouveau document Flash CS3, nous importons une image en bibliothèque, puis nous définissons une classe associée nommée `Logo`.

Nous instancions l'image et l'affichons :

```
// instantiation des données bitmaps  
var monLogo:Logo = new Logo(0,0);  
  
// creation d'une enveloppe Bitmap  
var monConteneur:Bitmap = new Bitmap ( monLogo );  
  
// ajout à l'affichage  
addChild ( monConteneur );  
  
// positionnement de l'image  
monConteneur.x = 100;  
monConteneur.y = 100;
```

La figure 12-15 montre le résultat :



*Figure 12-15. Affichage d'une image de bibliothèque.*

De manière générale, il n'est pas forcément nécessaire d'utiliser le type spécifique pour stocker l'instance de `BitmapData`. Nous pouvons aussi stocker l'instance de `Logo` au sein d'une variable de type `BitmapData` :

```
// instantiation des données bitmaps  
var monLogo:BitmapData = new Logo(0,0);
```

Si l'image en bibliothèque est un PNG transparent, l'instance de `BitmapData` créée est transparente :

```
// instantiation du logo  
var monLogo:BitmapData = new Logo(0,0);  
  
// affiche : true  
trace( monLogo.transparent );
```

Nous avons jusqu'à présent créé des images bitmaps de couleur unies, nous allons nous attarder maintenant à la modification des données bitmaps, en travaillant sur les pixels.

---

Il est important de noter qu'une image en bibliothèque ne possède pas de limitations de taille, contrairement aux instances de `BitmapData` créées par programmation.

---

## Peindre des pixels

Trois méthodes sont disponibles pour peindre les pixels d'une image, voici le détail de chacune d'entre elles :

- `BitmapData.setPixel` : peint un pixel au format RVB.
- `BitmapData.setPixel32` : peint un pixel au format ARVB.
- `BitmapData.setPixels` : peint des pixels d'après un tableau d'octets source définie par la classe `flash.utils.ByteArray`.

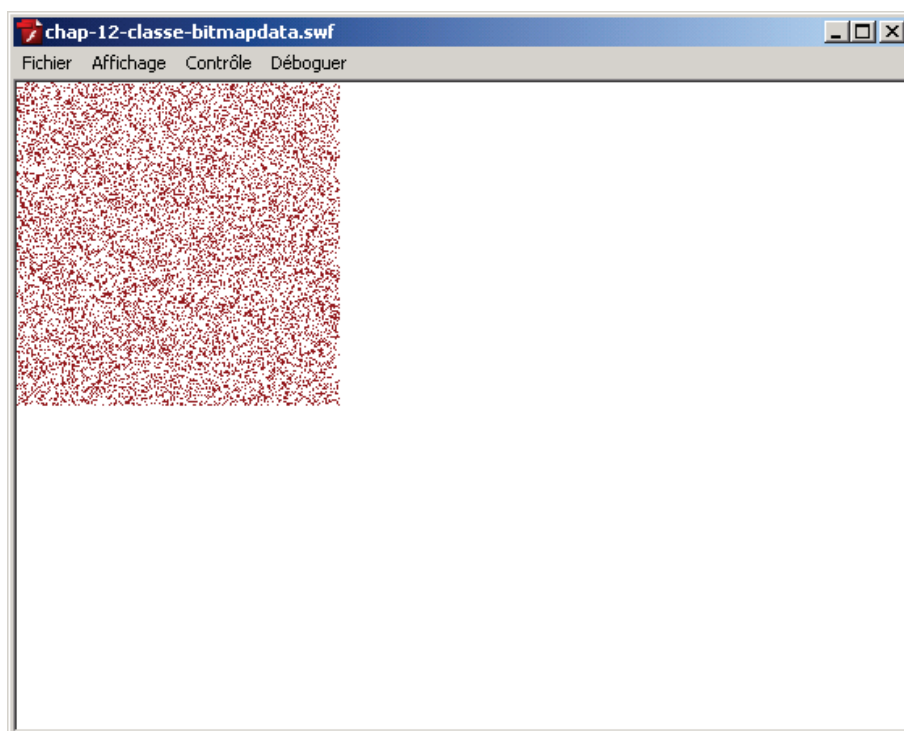
La méthode `setPixel` permet de colorer un pixel à une position donnée :

```
| public function setPixel(x:int, y:int, color:uint):void
```

Les paramètres `x` et `y` définissent la position du pixel à peindre. La couleur doit être spécifiée au format RVB. Dans le code suivant, nous nous colorons aléatoirement au sein d'une boucle certains pixels de l'image :

```
| var monImage:BitmapData = new BitmapData ( 200, 200, false, 0xFFFFFFFF );
|
| var conteneurImage:Bitmap = new Bitmap ( monImage );
|
| addChild ( conteneurImage );
|
| for ( var i:int = 0; i<20000; i++ )
| {
|     // positions aléatoires
|     // arrondi automatique, dû au type int
|     var positionX:int = Math.random()*251;
|     var positionY:int = Math.random()*251;
|
|     // peint un pixel
|     monImage.setPixel ( positionX, positionY, 0x990000 );
| }
| }
```

Le résultat est illustré en figure 12-16 :



*Figure 12-16. Dessin par `setPixel`.*

Si nous passons une couleur au format ARVB, le canal alpha est ignoré :

```
var monImage:BitmapData = new BitmapData ( 200, 200, false, 0xFFFFFFFF );  
  
var conteneurImage:Bitmap = new Bitmap ( monImage )  
  
addChild ( conteneurImage );  
  
for ( var i:int = 0; i<20000; i++ )  
{  
    // positions aléatoires  
    // arrondi automatique, dû au type int  
    var positionX:int = Math.random()*251;  
    var positionY:int = Math.random()*251;  
  
    // peint un pixel, le canal alpha est ignoré  
    monImage.setPixel ( positionX, positionY, 0x33990000 );  
}
```

En réalité lorsque nous appelons la méthode `setPixel` nous travaillons avec une couleur codée sur 24 bits, le canal alpha étant automatiquement défini.

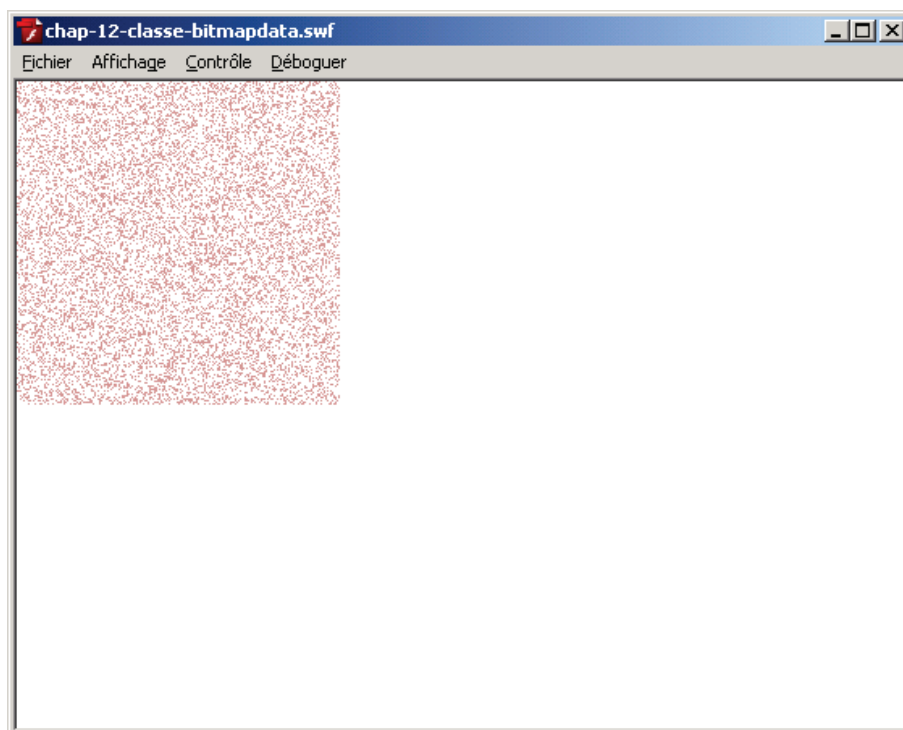
Afin de pouvoir peindre des pixels en précisant la transparence, nous devons utiliser la méthode `setPixel32` dont voici la signature :

```
| public function setPixel32(x:int, y:int, color:uint):void
```

Dans l'exemple suivant nous modifions la transparence de l'image en utilisant une couleur semi opaque :

```
var monImage:BitmapData = new BitmapData ( 200, 200, true, 0xFFFFFFFF );  
  
var conteneurImage:Bitmap = new Bitmap ( monImage )  
  
addChild ( conteneurImage );  
  
for ( var i:int = 0; i<20000; i++ )  
{  
    // positions aléatoires  
    // arrondi automatique, dû au type int  
    var positionX:int = Math.random()*251;  
    var positionY:int = Math.random()*251;  
  
    // peint les pixels avec une transparence de 40%  
    monImage.setPixel32 ( positionX, positionY, 0x66990000 );  
}
```

La figure 12-17 illustre le résultat :



*Figure 12-17. Dessin par setPixel32.*

Contrairement aux méthodes `setPixel` et `setPixel32` permettant de peindre un seul pixel par appel, la méthode `setPixels` permet de peindre une zone de pixels définie par une instance de la classe `flash.geom.Rectangle`.

Voici la signature de la méthode `setPixels` :

```
public function setPixels(rect:Rectangle, inputByteArray:ByteArray):void
```

Le premier paramètre accueille une instance de la classe `Rectangle` définissant la zone à peindre, puis en deuxième paramètre un tableau d'octets contenant la couleur de chaque pixel. Nous reviendrons sur la manipulation de données binaire au cours du chapitre 19 intitulé *ByteArray*.

Dans le code suivant, nous créons une image bitmap non transparente :

```
// création d'une image bitmap non transparente
var monImage:BitmapData = new BitmapData ( 200, 200, false, 0x99AAAA );

var conteneurImage:Bitmap = new Bitmap ( monImage )

addChild ( conteneurImage );
```

Puis un tableau binaire contenant la couleur de chaque pixel :

```
// tableau de pixels
var pixels:ByteArray = new ByteArray();

for ( var i:int = 0; i< 50; i++ )
{
    for ( var j:int = 0; j< 50; j++ )
    {
        // store la couleur de chaque pixel 32bits
        pixels.writeUnsignedInt(0x990000);
    }
}
```

Nous réinitialisons l'index de lecture du flux :

```
pixels.position = 0;
```

Puis, les pixels sont peints au sein de l'image bitmap :

```
monImage.setPixels( new Rectangle (0, 0, 50, 50), pixels );
```

La figure 12-18 illustre le résultat :

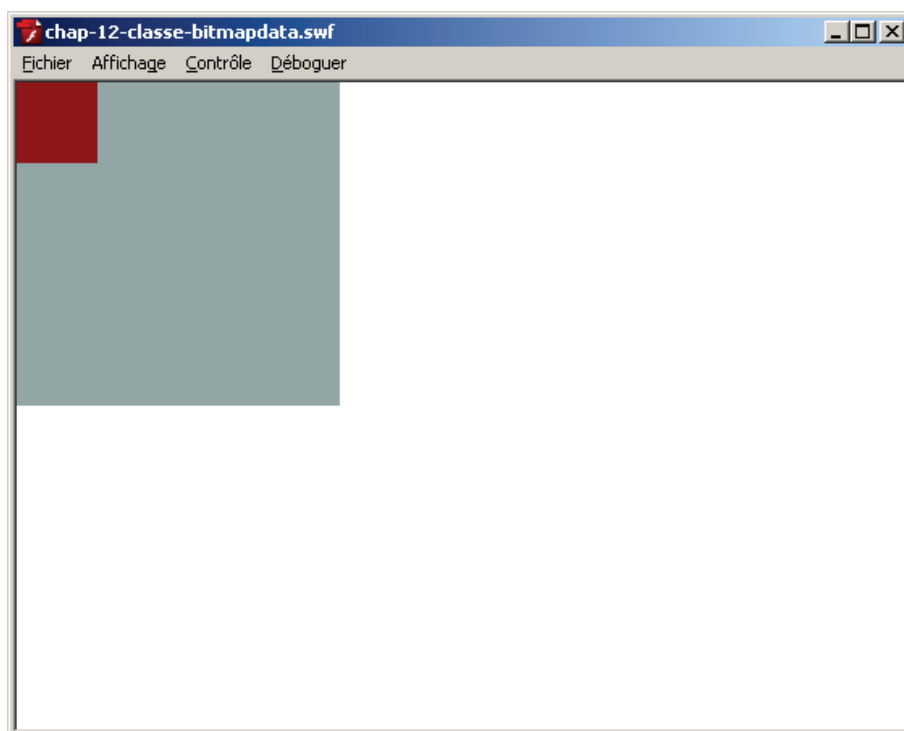


Figure 12-18. Dessin d'une zone par `setPixels`.

Bien que cette méthode puisse paraître pratique, elle ne s'avère pas être la plus efficace.

Nous sommes obligés de remplir un tableau d'octets contenant la couleur de chaque pixel puis d'appeler la méthode `setPixels`. Dans la plupart des cas nous préférons utiliser la méthode `setPixel` qui s'avère plus rapide.

## A retenir

- La méthode `setPixel` permet de peindre un pixel au format RVB.
- La méthode `setPixel32` permet de peindre un pixel au format ARVB.
- La méthode `setPixels` permet de peindre un ensemble de pixels, à partir d'un tableau d'octets source.

## Lire des pixels

Pour accéder à la couleur de chaque pixel nous disposons de trois autres méthodes dont voici le détail :

- `getPixel` : renvoie la couleur du pixel au format RVB selon un point spécifique.

- `getPixel32` : renvoie la couleur du pixel au format ARVB selon un point spécifique.
- `getPixels` : renvoie un tableau d'octets contenant les couleurs de pixels défini selon une zone rectangulaire (`flash.geom.Rectangle`)

Afin de lire les pixels nous pouvons utiliser les méthodes correspondantes `getPixel` et `getPixel32`. La méthode `getPixel` renvoie la couleur du pixel au format RVB et possède la signature suivante :

```
public function getPixel(x:int, y:int):uint
```

Nous allons créer une image de couleur verte et récupérer la couleur d'un pixel :

```
var monImage:BitmapData = new BitmapData ( 200, 200, false, 0xFFFF00 );  
  
var conteneurImage:Bitmap = new Bitmap ( monImage );  
  
addChild ( conteneurImage );  
  
var couleurPixel:Number = monImage.getPixel( 0, 0 );  
  
// affiche : FFFF00  
trace( couleurPixel.toString(16).toUpperCase() );
```

Si nous tentons de récupérer la couleur d'un pixel composant une image transparente, le canal alpha est ignoré :

```
var monImage:BitmapData = new BitmapData ( 200, 200, true, 0x99FF0088 );  
  
var conteneurImage:Bitmap = new Bitmap ( monImage );  
  
addChild ( conteneurImage );  
  
var couleurPixel:Number = monImage.getPixel ( 0, 0 );  
  
// affiche : FF0088  
// le canal alpha est ignoré  
trace( couleurPixel.toString(16).toUpperCase() );
```

La méthode `getPixel32` permet, de récupérer la couleur d'un pixel au format ARVB :

```
var monImage:BitmapData = new BitmapData ( 200, 200, true, 0x99FF0088 );  
  
var conteneurImage:Bitmap = new Bitmap ( monImage );  
  
addChild ( conteneurImage );  
  
var couleurPixel:Number = monImage.getPixel32 ( 0, 0 );  
  
// affiche : 99FF0088  
trace( couleurPixel.toString(16).toUpperCase() );
```



Alors que les méthodes `getPixel` et `getPixel32` existent depuis le lecteur Flash 8, la méthode `getPixels` a été introduite par ActionScript 3.

Celle-ci a l'avantage de renvoyer un tableau binaire contenant les pixels de la zone spécifiée. En ActionScript 1 et 2, nous étions obligés de parcourir manuellement l'image bitmap afin d'obtenir l'ensemble des pixels.

Dans le code suivant nous examinons une image bitmap et stockons chaque pixel dans un tableau :

```
// instantiation du logo
var monLogo:Logo = new Logo(0,0);

// affichage
var monConteneur:Bitmap = new Bitmap ( monLogo );

// ajout à l'affichage
addChild ( monConteneur );

// positionnement de l'image
monConteneur.x = 100;
monConteneur.y = 100;

// récupération largeur et hauteur
var largeur:Number = monConteneur.width;
var hauteur:Number = monConteneur.height;

// tableau contenant les pixels
var tableauPixels:Array = new Array();

for ( var i:int = 0; i< largeur; i++ )
{
    for ( var j:int = 0; j< hauteur; j++ )
    {
        // récupère la couleur de chaque pixel
        var couleur:Number = monLogo.getPixel ( i, j );

        // stocke chaque couleur au sein d'un tableau
        tableauPixels.push ( couleur );
    }
}

// affiche : 17424
trace( tableauPixels.length );
```

En affichant la longueur du tableau, nous voyons que 17424 pixels sont stockés au sein du tableau. Cette opération fonctionnait sans problème sur des images bitmapss de petite taille. Pour des images de dimensions élevées, l'utilisation de boucles imbriquées n'était plus possible.

En ActionScript 3, nous utilisons la méthode `getPixels` :

```
// instantiation du logo
var monLogo:Logo = new Logo(0,0);

// affiche : 132, 132 (132*132 = 17424)
trace( monLogo.width, monLogo.height );

var tableauPixels:ByteArray = monLogo.getPixels ( monLogo.rect );

// affiche : 69696
trace( tableauPixels.length );
```

Celle-ci retourne un tableau d'octets contenant les pixels de la zone spécifiée. Au sein d'un tableau d'octets, un pixel occupe 4 index. Si nous divisons 69696 par 4 nous obtenons 17424 pixels.

L'utilisation de la méthode `getPixels` s'avère beaucoup plus rapide que la méthode `getPixel`, car la totalité des pixels d'une image peut être retournée instantanément.

## A retenir

- La méthode `getPixel` retourne la couleur d'un pixel au format RVB.
- La méthode `getPixel32` retourne la couleur d'un pixel au format ARVB.
- La méthode `getPixels` retourne un tableau d'octets contenant un ensemble de pixels.

## Accrochage aux pixels

Lorsqu'une image bitmap est affichée nous pouvons garantir l'accrochage aux pixels grâce au paramètre `pixelSnapping`.

Trois constantes sont utilisées afin de déterminer l'accrochage d'une image :

- `PixelSnapping.ALWAYS` : l'image est toujours accrochée au pixel le plus proche.
- `PixelSnapping.AUTO` : l'image bitmap est accrochée au pixel le plus proche si elle est dessinée sans rotation ni inclinaison et que son facteur de redimensionnement est compris entre 99,9 % et 100,1 %.
- `PixelSnapping.NEVER` : l'accrochage aux pixels est désactivé.

Par défaut la valeur du paramètre est à `auto` :

```
Bitmap(bitmapData:BitmapData = null, pixelSnapping:String = "auto",
smoothing:Boolean = false)
```

En cas de rotation ou transformation de l'image affichée, celle-ci pourrait voir ses coordonnées glisser sur des coordonnées flottantes,

donnant un aspect flouté aux contours de l'image. L'accrochage au pixels garantie un rendu net de l'image affichée.

## Le lissage

Grâce au lissage, la classe `Bitmap` permet d'améliorer le rendu d'une image lorsque celle-ci est redimensionnée.

Nous avons la possibilité d'activer le lissage grâce au dernier paramètre du constructeur de la classe `Bitmap` :

```
// instantiation du logo
var monLogo:Logo = new Logo(0,0);

// affichage de l'image en accrochant toujours les pixels et en désactivant le lissage
var monConteneur:Bitmap = new Bitmap ( monLogo, PixelSnapping.ALWAYS, false );

// ajout à l'affichage
addChild ( monConteneur );

// positionnement de l'image
monConteneur.x = 100;
monConteneur.y = 100;
// redimensionnement
monConteneur.scaleX = monConteneur.scaleY = 1.2;

// affichage de l'image en accrochant toujours les pixels et en activant le lissage
var monConteneurBis:Bitmap = new Bitmap ( monLogo, PixelSnapping.ALWAYS, true );
// ajout à l'affichage
addChild ( monConteneurBis );

// positionnement de l'image
monConteneurBis.x = 300;
monConteneurBis.y = 100;

// redimensionnement
monConteneurBis.scaleX = monConteneurBis.scaleY = 1.2;
```

La figure 12-19 illustre la différence entre une image non lissée et lissée :



*Figure 12-19. Images bitmaps non lissée et lissée.*

En adoucissant l'interpolation des pixels, l'image conserve un rendu lissé lorsque celle-ci doit être redimensionnée.

## Mise en cache des bitmap à l'exécution

La mise en cache des bitmap à l'exécution est une fonctionnalité accessible par programmation ou depuis l'environnement auteur, permettant d'accélérer grandement la vitesse de rendu d'un élément vectoriel. Celle-ci est exclusivement réservée aux objets de type `DisplayObject`. Pour comprendre cette fonctionnalité, nous allons nous attarder quelques instants sur le système de rendu du lecteur Flash.

Le terme de mise en cache illustre le mécanisme interne du lecteur visant à créer temporairement en mémoire une version bitmap de l'objet vectoriel. En activant la mise en cache des bitmap sur un `DisplayObject` de 300 \* 300 pixels, une image bitmap 32 bits de même taille est créée en mémoire puis affichée en remplacement de l'objet vectoriel. Cette technique permet au lecteur d'afficher simplement l'image stockée en mémoire et de ne plus rendre les données vectorielles, entraînant ainsi une augmentation significative de la vitesse d'affichage.

Pour mettre en cache un objet graphique, il suffit d'activer la propriété `cacheAsBitmap` de la classe `DisplayObject` :

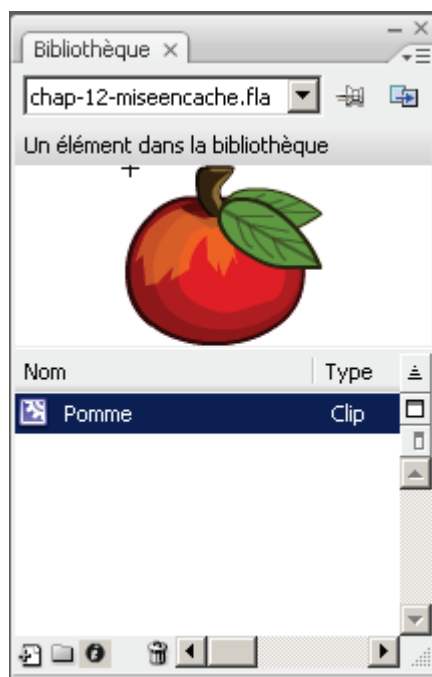
```
DisplayObject.cacheAsBitmap = true;
```

Pour désactiver la mise en cache, nous passons la valeur booléenne `false` à la propriété `cacheAsBitmap` :

```
DisplayObject.cacheAsBitmap = false;
```

Lorsque la mise en cache est désactivée, le bitmap associé est automatiquement supprimé de la mémoire. Afin de mettre en avant l'intérêt de la mise en cache des bitmap à l'exécution nous allons mettre cette fonctionnalité en pratique.

Dans un nouveau document Flash CS3 nous créons un nouveau symbole clip représentant une pomme. La figure 12-20 illustre le symbole utilisé :



*Figure 12-20. Symbole Pomme.*

Grâce au panneau *Liaison* nous associons une classe nommée *Pomme* que nous définissons à côté du document Flash en cours. Celle-ci contient le code suivant :

```
package
{
    import flash.display.MovieClip;
    import flash.events.Event;

    public class Pomme extends MovieClip
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function Pomme ()
        {
            addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );
            addEventListener ( Event.REMOVED_FROM_STAGE, supprimeAffichage );
        }

        public function ajoutAffichage ( pEvt:Event ):void
        {
            init();
        }
    }
}
```

```
        addEventListener ( Event.ENTER_FRAME, mouvement );
    }

    private function supprimerAffichage ( ):void
    {
        removeEventListener ( Event.ENTER_FRAME, mouvement );
    }

    private function init ( ):void
    {
        destinationX = Math.random()*(stage.stageWidth-width);
        destinationY = Math.random()*(stage.stageHeight-height);
    }

    private function mouvement ( pEvt:Event ):void
    {
        x -= ( x - destinationX ) *.5;
        y -= ( y - destinationY ) *.5;

        if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y -
destinationY ) < 1 ) init();
    }
}
}
```

Puis nous affichons différentes instances du symbole **Pomme** :

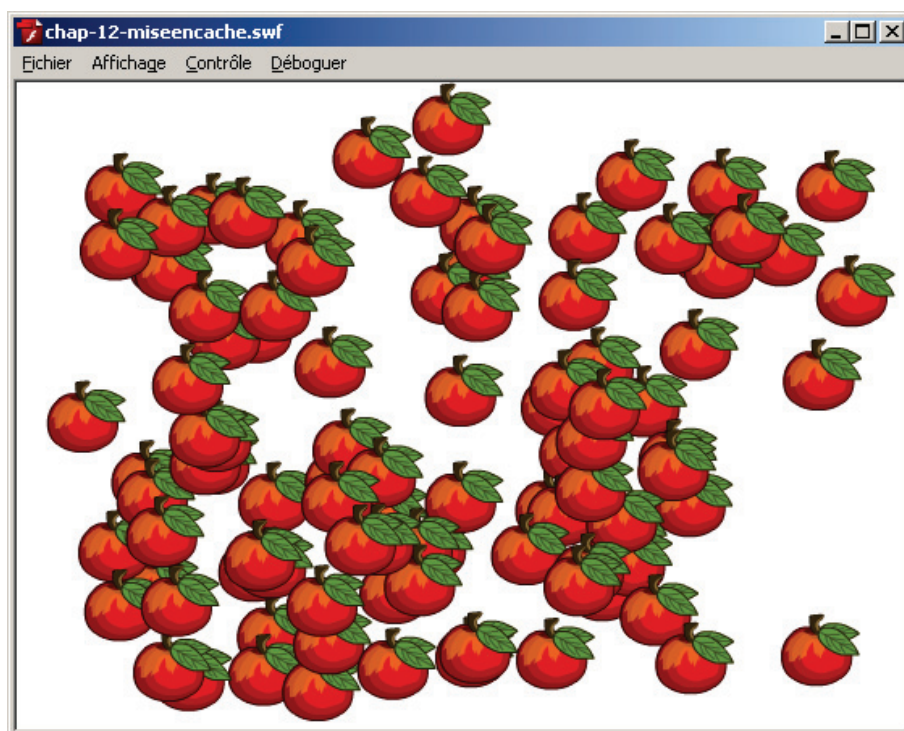
```
var conteneur:Sprite = new Sprite();

addChild ( conteneur );

for ( var i:int = 0; i< 100; i++ )
{
    var maPomme:Pomme = new Pomme();

    conteneur.addChild ( maPomme );
}
```

Chaque instance se déplace aléatoirement sur la scène comme l'illustre la figure 12-21 :



*Figure 12-21. Déplacement des pommes.*

Nous remarquons que l'animation n'est pas très fluide. Nous allons optimiser le rendu en activant la mise en cache des bitmaps à l'exécution :

```
var conteneur:Sprite = new Sprite();
addChild ( conteneur );

for ( var i:int = 0; i < 100; i++ )
{
    var maPomme:Pomme = new Pomme();
    conteneur.addChild ( maPomme );
}

stage.addEventListener ( MouseEvent.CLICK, miseEnCache );

function miseEnCache ( pEvt:MouseEvent ):void
{
    var lng:int = conteneur.numChildren;

    for ( var i:int = 0; i < lng; i++ )
    {
        var pomme:Pomme = Pomme ( conteneur.getChildAt ( i ) );
        pomme.cacheAsBitmap = ! Boolean ( pomme.cacheAsBitmap );
    }
}
```

```
    }  
}
```

Lorsque nous cliquons sur la scène, nous activons ou désactivons la mise en cache des bitmap sur chaque pomme, le rendu est grandement accéléré.

L'utilisation de cette fonctionnalité entraîne les mêmes considérations que la création d'images bitmaps avec la classe `BitmapData`. Chaque pomme mesure 47,5 \* 43 pixels, cela correspond pour chaque pomme à une image bitmap de 7,97 Ko en mémoire. Nous affichons dans le code précédent 100 pommes, l'activation de la mise en cache des bitmap consomme donc pour cette animation 797 Ko en mémoire vive.

Ainsi, il convient de veiller aux dimensions de l'objet mis en cache. Un élément vectoriel de plus de 2880 pixels ne peut être mis en cache car la création d'une telle image bitmap en mémoire est impossible pour les raisons évoquées en début de chapitre. Cette fonctionnalité de mise en cache des bitmap peut paraître comme la situation à un grand nombre de problèmes liés aux performances, mais il faut prendre en considération certains effets pervers souvent méconnus pouvant inverser la donne.

## A retenir

- Afin d'activer la mise en cache des bitmap à l'exécution, nous passons la valeur `true` à la propriété `cacheAsBitmap`.
- Afin de désactiver la mise en cache des bitmap à l'exécution, nous passons la valeur `false` à la propriété `cacheAsBitmap`.
- L'intérêt de la mise en cache des bitmap, consiste à afficher une représentation bitmap de l'objet vectoriel.
- La mise en cache des bitmap augmente sensiblement la vitesse de rendu mais requiert plus de mémoire.
- Lorsque la mise en cache des bitmaps est désactivée, l'image bitmap associée est supprimée de la mémoire.

## Effets pervers

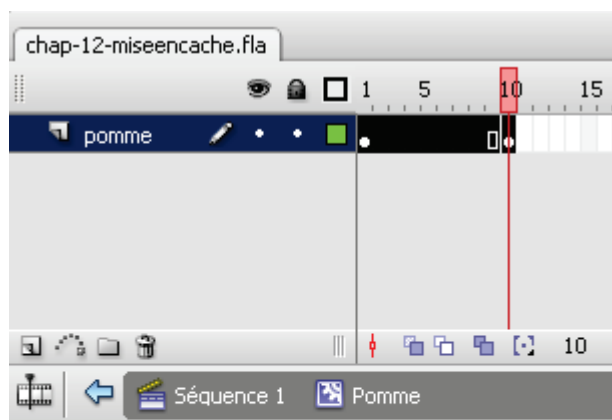
La mise en cache des bitmap à l'exécution est une fonctionnalité qui doit être utilisée avec réflexion. Si celle-ci n'est pas maîtrisée, nous obtenons l'inverse du résultat escompté.

Lorsqu'un `DisplayObject` subit une transformation autre qu'une simple translation en x et y, la mise en cache des bitmap est à éviter. Chaque étirement, rotation, changement d'opacité, ou déplacement de



la tête de lecture nécessite une mise à jour de l'image bitmap créée en mémoire.

Afin de mettre en évidence cet effet pervers, nous ajoutons une image clé à l'image 10 du symbole **Pomme** comme l'illustre la figure 12-22 :



*Figure 12-22. Agrandissement du symbole **Pomme**.*

Sur cette image clé, nous agrandissons la taille de la pomme. Si nous testons à nouveau notre animation et activons la mise en cache des bitmap. Nous remarquons qu'à chaque passage de la tête de lecture sur l'image 10, le lecteur détecte un changement de taille et met à jour l'image bitmap en cache. Ce processus ralentit grandement l'affichage et annule l'intérêt de la fonctionnalité.

De la même manière, si nous procédons simplement à une rotation de chaque instance, le lecteur met à jour le bitmap pour chaque nouvelle image. Nous modifions la méthode **mouvement** au sein de la classe **Pomme** :

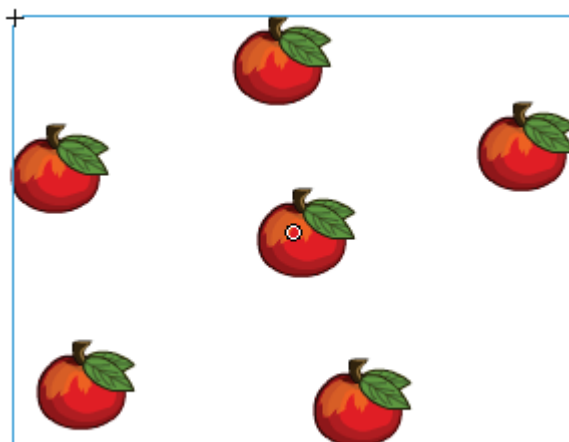
```
private function mouvement ( pEvt:Event ):void
{
    rotation += 5;

    x -= ( x - destinationX ) *.5;
    y -= ( y - destinationY ) *.5;

    if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY ) < 1 )
    {
        init();
    }
}
```

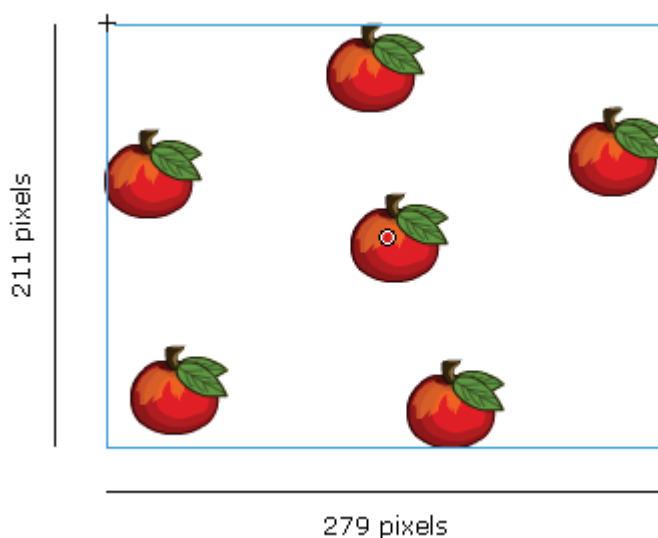
Pour chaque nouvelle image parcourue, le lecteur Flash met à jour le bitmap associé. Il convient donc d'utiliser la mise en cache des bitmap uniquement lorsque l'objet subit une translation en x et y.

Nous allons nous intéresser maintenant à un autre effet pervers. La figure 12-23 illustre un symbole clip contenant plusieurs instances du symbole *Pomme* :



*Figure 12-23. Clip contenant différentes instances du symbole *Pomme*.*

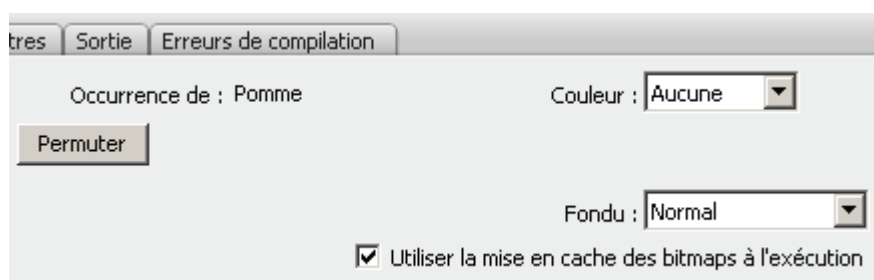
Afin de gagner du temps, nous pourrions être tentés d'activer la mise en cache sur le clip conteneur. Cela entraîne la création d'une image bitmap transparente en mémoire de la taille du conteneur comme l'illustre la figure 12-24 :



*Figure 12-24. Dimensions du clip conteneur.*

La mise en cache du clip conteneur crée un bitmap de 229,95 Ko en mémoire. Toute la surface transparente est stockée en mémoire inutilement. Il serait plus judicieux d'activer la mise en cache sur chaque instance du symbole **Pomme**, ce qui nécessiterait 55,79 Ko en mémoire.

Il est aussi possible d'activer la mise en cache des bitmap à l'exécution au sein de l'environnement auteur en sélectionnant l'objet graphique à mettre en cache puis en cochant la case correspondante au sein de l'inspecteur de propriétés :

*Figure 12-25. Mise en cache des bitmaps à l'exécution depuis l'environnement auteur.*

Il convient d'utiliser cette fonctionnalité avec attention, de plus l'utilisation de filtres est directement liée à la mise en cache des bitmaps à l'exécution. Nous allons nous y intéresser à présent afin de mieux comprendre le fonctionnement des filtres.

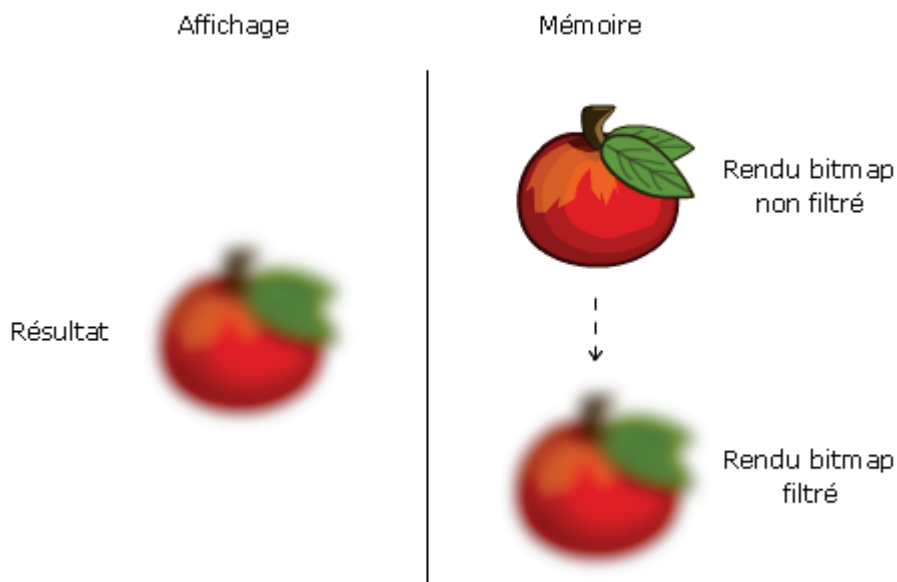
## A retenir

- La mise en cache des bitmaps à l'exécution doit être activée uniquement sur des objets subissant une translation en x et y.
- Le lecteur met à jour le bitmap associé pour toute modification.
- Si cette fonctionnalité n'est pas maîtrisée, nous obtenons un ralentissement de la vitesse de rendu et une forte occupation mémoire.

## Filtrer un élément vectoriel

Les filtres sont intimement liés à la notion de bitmap. Lorsque nous utilisons un filtre sur un objet vectoriel, le lecteur Flash crée en mémoire deux images bitmaps afin de produire le résultat filtré.

La figure 12-26 illustre le mécanisme interne :



*Figure 12-26. Mécanisme de création de filtres.*

Le premier bitmap est utilisé pour représenter l'objet non filtré, en réalité le lecteur utilise la mise en cache des bitmaps à l'exécution afin de générer un premier bitmap sur lequel travailler pour appliquer le filtre. Le deuxième bitmap sert à accueillir le rendu filtré. L'utilisation de filtres requiert donc deux fois plus de mémoire que la mise en cache des bitmap à l'exécution et entraîne les mêmes précautions d'utilisation.

Afin d'affecter un filtre à un `DisplayObject`, nous affectons un tableau de filtres à la propriété `filters`. L'utilisation d'un tableau permet de cumuler plusieurs filtres appliqués à un objet graphique et de modifier la superposition de chaque filtre.

Dans un nouveau document Flash CS3, nous posons une instance du symbole `Pomme` sur la scène et lui donnons comme nom d'occurrence `pomme` :

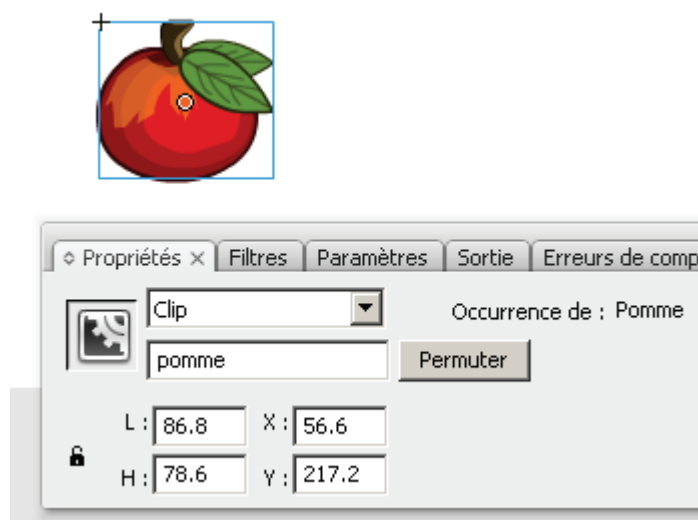


Figure 12-27. Occurrence du symbole *Pomme*.

Puis nous ajoutons dynamiquement un des filtres situés dans le paquetage `flash.filters`. Voici en détail les différents filtres disponibles :

- `flash.filters.BevelFilter` : applique un effet de biseau.
- `flash.filters.GradientBevelFilter` : applique un effet de biseau dégradé.
- `flash.filters.BlurFilter` : applique un effet de flou.
- `flash.filters.GlowFilter` : applique un effet de rayonnement.
- `flash.filters.GradientGlowFilter` : applique un effet de rayonnement dégradé.
- `flash.filters.ColorMatrixFilter` : applique une transformation de couleurs à chaque pixel.
- `flash.filters.ConvolutionFilter` : applique un filtre de convolution de matrice.
- `flash.filters.DisplacementMapFilter` : applique un effet de déplacement sur chaque pixel.
- `flash.filters.DropShadowFilter` : applique un effet d'ombre portée.

Nous allons utiliser la classe `BlurFilter` pour affecter un filtre de flou, dont voici le constructeur :

```
public fonction BlurFilter(blurX:Number = 4.0, blurY:Number = 4.0, quality:int = 1)
```

Les deux premiers paramètres concernent la dilatation des pixels pour chaque axe. Le dernier paramètre permet de spécifier la qualité du

résultat final, il s'agit en réalité du nombre de passage du filtre. Une valeur comprise entre 1 et 3 est généralement utilisée.

---

Quelque soit la qualité du filtre ou dilatation des pixels, le poids des images bitmaps créées en mémoire reste le même. A l'inverse, la vitesse d'affichage est fortement liée à la dilatation des pixels ainsi que la qualité. Pour des raisons de performances il est fortement recommandé de toujours utiliser des multiples de 2 pour les quantités de flous et de ne jamais dépasser une qualité de 15.

---

Dans le code suivant nous ajoutons un filtre de flou :

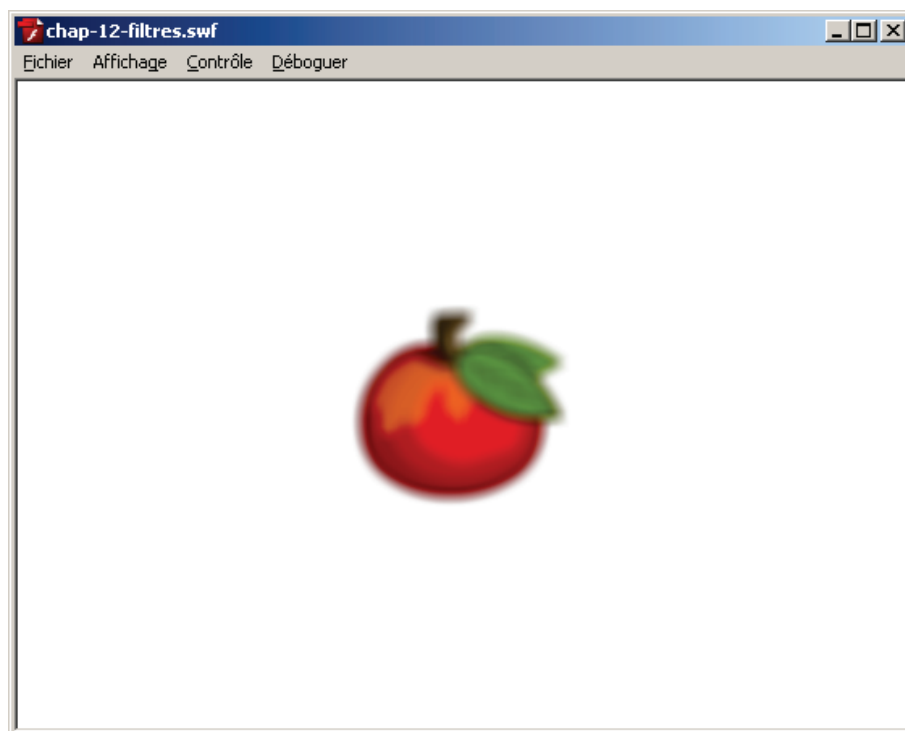
```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter (10, 10, 1);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;
```

La figure 12-28 illustre le résultat :



*Figure 12-28. Filtre de flou.*

Trois qualités de filtres sont disponibles et accessible depuis des constantes de la classe `BitmapFilterQuality` :

- `BitmapFilterQuality.LOW` : qualité inférieure.
- `BitmapFilterQuality.MEDIUM` : qualité moyenne.
- `BitmapFilterQuality.HIGH` : qualité supérieure, s'approche du flou gaussien.

Il est donc possible de spécifier manuellement la qualité du filtre, mais pour des raisons de portabilité nous préférons toujours utiliser des constantes de classe :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;
```

Une fois le filtre appliqué nous remarquons que la mise en cache des bitmaps est activée automatiquement :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// affiche : true
trace( pomme.cacheAsBitmap );
```

L'instance du symbole `Pomme` mesure 122 \* 110 pixels. Dans notre exemple, le filtre de flou pèse en mémoire 104,84 Ko.

Afin de faciliter le calcul du poids d'un objet en mémoire contenant différents filtres nous pouvons ajouter au sein de notre classe `BitmapOutils` une nouvelle méthode appelée `poidsFiltres` :

```
package org.ouutils
{
    import flash.display.BitmapData;
    import flash.display.DisplayObject;
```

```
public class BitmapOutils
{
    public static function hexArgb ( pCouleur:Number ):Object
    {
        var composants:Object = new Object();
        composants.alpha = (pCouleur >>> 24) & 0xFF;
        composants.rouge = (pCouleur >>> 16) & 0xFF;
        composants.vert  = (pCouleur >>> 8) & 0xFF;
        composants.bleu   = pCouleur & 0xFF;

        return composants;
    }

    public static function argbHex ( pAlpha:int, pRouge:int, pVert:int,
    pBleu:int ):uint
    {
        return ( pAlpha << 24 | pRouge << 16 | pVert << 8 | pBleu );
    }

    public static function poids ( pBitmapData:BitmapData ):Number
    {
        return (pBitmapData.width * pBitmapData.height) * 4;
    }

    public static function poidsFiltres ( pDisplayObject:DisplayObject
    ):Number
    {
        return ((pDisplayObject.width * pDisplayObject.height) * 4) *
        pDisplayObject.filters.length) * 2;
    }
}
```

Cette méthode nous permet de connaître le poids total d'un objet filtré en mémoire :

```
import org.ouutils.BitmapOutils;

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );
```



```
// affectation du filtre
pomme.filters = filtresEnCours;

// affiche : 112.08052734375
trace( BitmapOutils.poidsFiltres ( pomme ) / 1024 );
```

L'instance du symbole `Pomme` pèse en mémoire environ 112 Ko. Si nous souhaitons ajouter de nouveaux filtres il n'est pas possible d'ajouter directement un filtre au tableau `filters` :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );
```

Nous devons obligatoirement affecter à nouveau à la propriété `filters` un tableau contenant la totalité des filtres :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

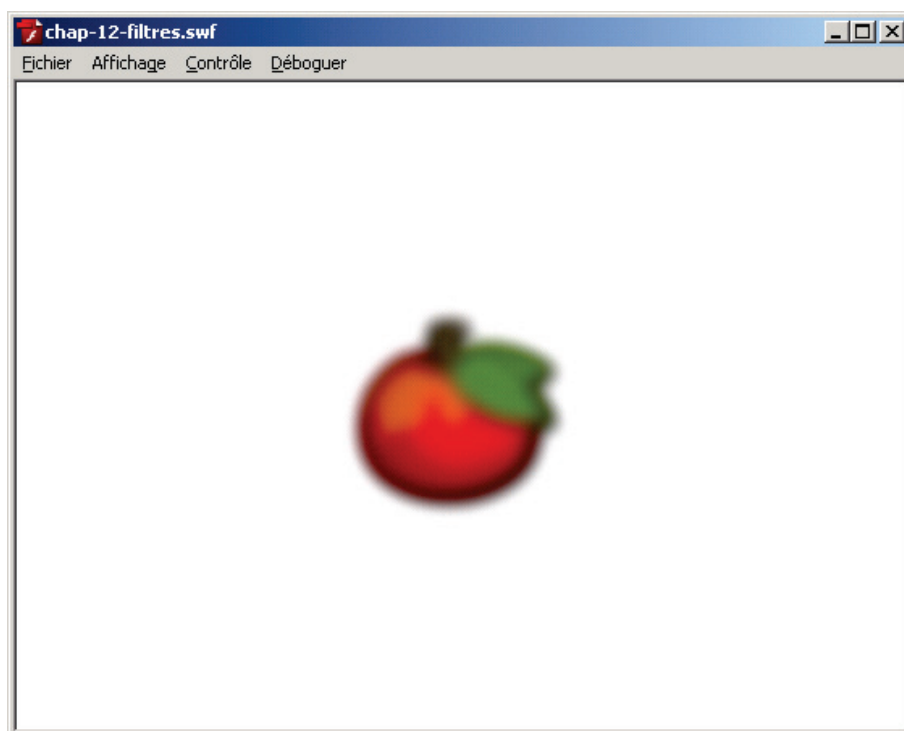
// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// affectation des filtres
pomme.filters = filtresEnCours;
```

Le code précédent génère le résultat suivant :



*Figure 12-29. Filtre de flou et ombre portée.*

Si nous évaluons à nouveau le poids de l'instance du symbole **Pomme** en mémoire, nous remarquons que son poids en mémoire a doublé et nécessite désormais 224 Ko en mémoire :

```
import org.ouutils.BitmapOutils;

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// affectation des filtres
pomme.filters = filtresEnCours;

// affiche : 224.1610546875
trace( BitmapOutils.poidsFiltres ( pomme ) / 1024 );
```

Il n'existe aucune méthode native permettant d'inverser la position des filtres au sein du tableau `filters`. Si nous souhaitons supprimer un filtre, nous devons travailler avec les méthodes de classe `Array` puis affecter à nouveau le tableau de filtres modifié. Dans le code suivant nous supprimons le filtre de flou :

```
import org.ouutils.BitmapOutils;

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

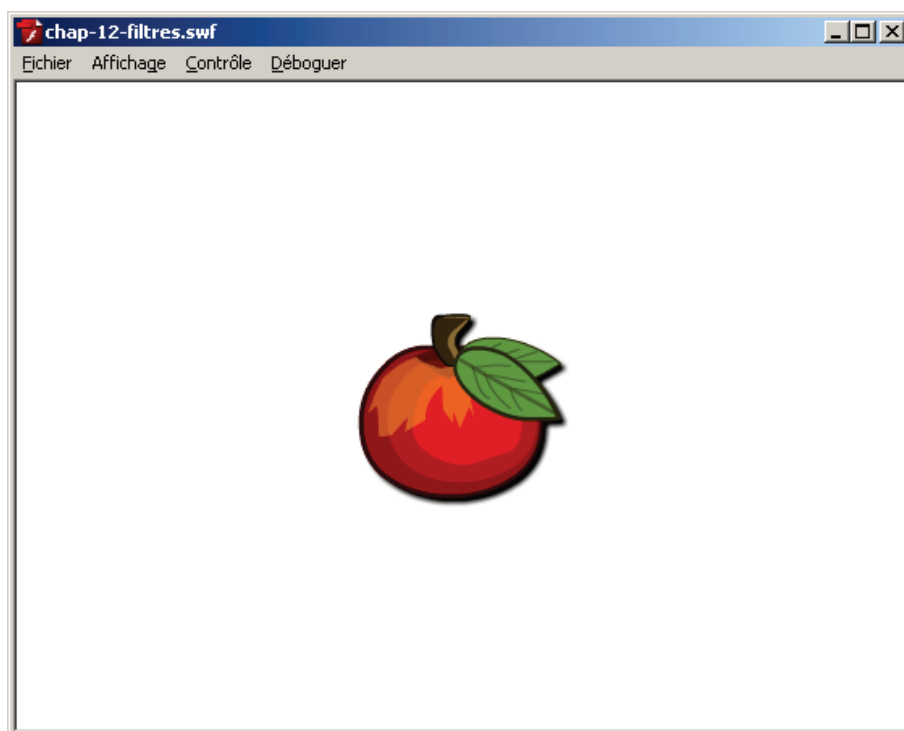
// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// suppression du filtre de flou
filtresEnCours.splice ( 0, 1 );

// affectation des filtres
pomme.filters = filtresEnCours;

// affiche : 112.08052734375
trace( BitmapOutils.poidsFiltres ( pomme ) / 1024 );
```

La figure 12-30 illustre le résultat :



*Figure 12-30. Ombre portée.*

A l'aide de la méthode `splice` nous avons supprimé le filtre de flou positionné à l'index 0. Dans certaines situations, si nous ne connaissons pas la position d'un filtre au sein du tableau interne, nous pouvons utiliser la méthode `indexOf` de la classe `Array` :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// récupère la position du filtre
var position:int = filtresEnCours.indexOf( filtreFlou );

// si le filtre est trouvé
if ( position != -1 )
{
```

```
        // le filtre est supprimé
        filtresEnCours.splice ( position, 1 );

    } else trace ( "Filtre non présent !" );

    // affectation des filtres
    pomme.filters = filtresEnCours;
```

Grâce à la méthode `indexOf`, nous n'avons pas besoin de savoir la position du filtre dans le tableau interne. L'index retourné nous permet de supprimer le filtre correspondant.

Il n'existe pas de méthode `dispose` pour libérer la mémoire utilisée par un filtre. Si nous souhaitons libérer les ressources nous devons supprimer les références pointant vers le filtre. Dans le code suivant, nous rendons le filtre de flou éligible à la suppression par le *ramasse-miettes* :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// récupère la position du filtre
var position:int = filtresEnCours.indexOf( filtreFlou );

// si le filtre est trouvé
if ( position != -1 )
{
    // le filtre est supprimé
    filtresEnCours.splice ( position, 1 );
    // puis désactivé
    filtreFlou = null;
} else trace ( "Filtre non présent !" );

// affectation des filtres
pomme.filters = filtresEnCours;
```

Afin de supprimer la totalité des filtres affectés à un `DisplayObject`, nous affectons à la propriété `filters` un tableau vide :

```
| // création du filtre de flou
```

---

```
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

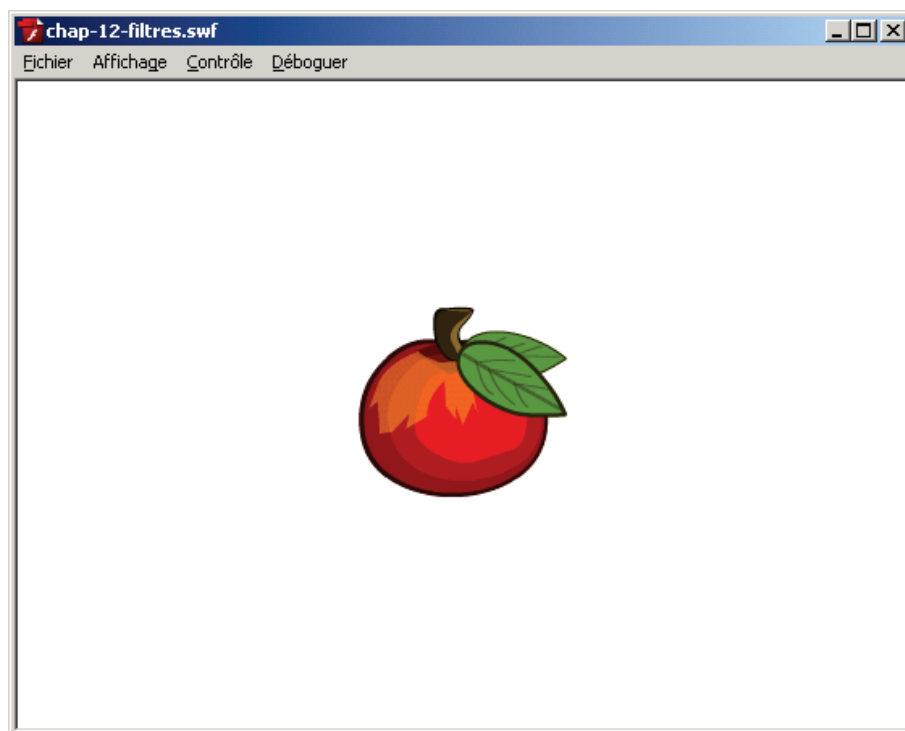
// affectation des filtres
pomme.filters = filtresEnCours;

// écrase le tableau de filtres existants
filtresEnCours = new Array();

// désactivation des filtres
filtreFlou = null;
ombrePortee = null;

// mise à jour de l'affichage
pomme.filters = filtresEnCours;
```

La figure 12-31 illustre le résultat final :



*Figure 12-31. Suppression des filtres.*

Une fois que les filtres ne sont plus liés au `DisplayObject`, la mise en cache des bitmaps à l'exécution est désactivée automatiquement :

```
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// tableau de filtres
var filtresEnCours:Array = new Array();

// ajout du filtre de flou
filtresEnCours.push ( filtreFlou );

// affectation du filtre
pomme.filters = filtresEnCours;

// création d'une ombre portée
var ombrePortee:DropShadowFilter = new DropShadowFilter ();

// ajout du filtre d'ombre portée
filtresEnCours.push ( ombrePortee );

// affectation des filtres
pomme.filters = filtresEnCours;

// écrase le tableau de filtres existants
filtresEnCours = new Array();

// désactivation des filtres
filtreFlou = null;
ombrePortee = null;

// mise à jour de l'affichage
pomme.filters = filtresEnCours;

// affiche : false
trace( pomme.cacheAsBitmap );
```

Attention, la désactivation des filtres en mémoire par suppression des références repose sur l'intervention du *ramasse-miettes*.

---

## A retenir

---

- L'utilisation de filtres sur un `DisplayObject` active automatiquement la mise en cache des bitmaps à l'exécution.
- En plus du premier bitmap créé lors de la mise en cache des bitmaps à l'exécution, un deuxième est créé afin de produire le rendu filtré.
- Les mêmes précautions d'utilisation liées à l'activation de la mise en cache des bitmaps doivent être appliquées lors de l'utilisation de filtres appliqués à des éléments vectoriels.
- La désactivation des filtres s'appuie sur l'intervention du *ramasse-miettes*.

## Filtrer une image bitmap

Il est aussi possible d'appliquer différents filtres à une image bitmap. Pour cela nous utilisons la méthode `applyFilter` de la classe `BitmapData` dont voici la signature :

```
public function applyFilter(sourceBitmapData:BitmapData, sourceRect:Rectangle,
    destPoint:Point, filter:BitmapFilter):void
```

Celle ci accepte quatre paramètres :

- `sourceBitmapData` : les données bitmaps à filtrer.
- `sourceRect` : la zone sur laquelle le filtre est appliqué.
- `destPoint` : point de départ utilisé par le rectangle.
- `Filter` : le filtre à appliquer.

Dans le code suivant, nousinstancions une image provenant de la bibliothèque. Un filtre de flou est partiellement appliqué :

```
// instantiation du logo
var donneesBitmap:Logo = new Logo ( 0, 0 );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

monImage.x = (stage.stageWidth - monImage.width) / 2;
monImage.y = (stage.stageHeight - monImage.height) / 2

addChild ( monImage );

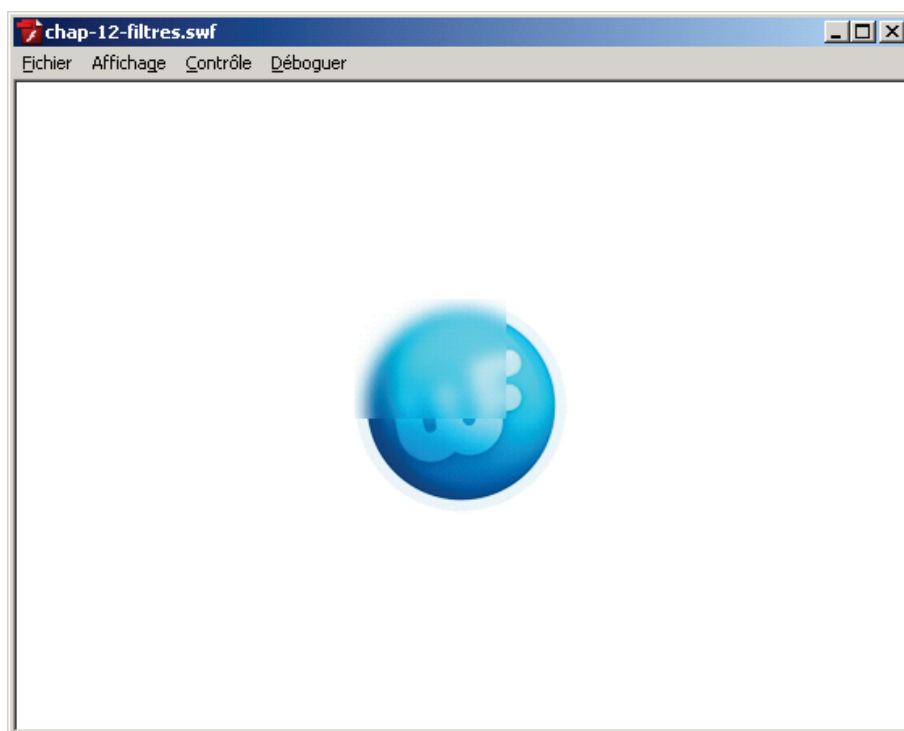
// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH
);

// définition d'une surface
var zone:Rectangle = new Rectangle ( 0, 0, 80, 60 );

// affectation du filtre de flou sur une surface limitée
donneesBitmap.applyFilter( donneesBitmap, zone, new Point(0,0), filtreFlou );
```

La figure 12-32 illustre le résultat :





*Figure 12-32. Filtre partiel.*

Afin d'affecter le filtre sur la surface totale de l'image, nous passons l'objet `Rectangle` accessible par la propriété `rect` de la classe `BitmapData` :

```
// instantiation du logo
var donneesBitmap:Logo = new Logo ( 0, 0 );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

monImage.x = (stage.stageWidth - monImage.width) / 2;
monImage.y = (stage.stageHeight - monImage.height) / 2

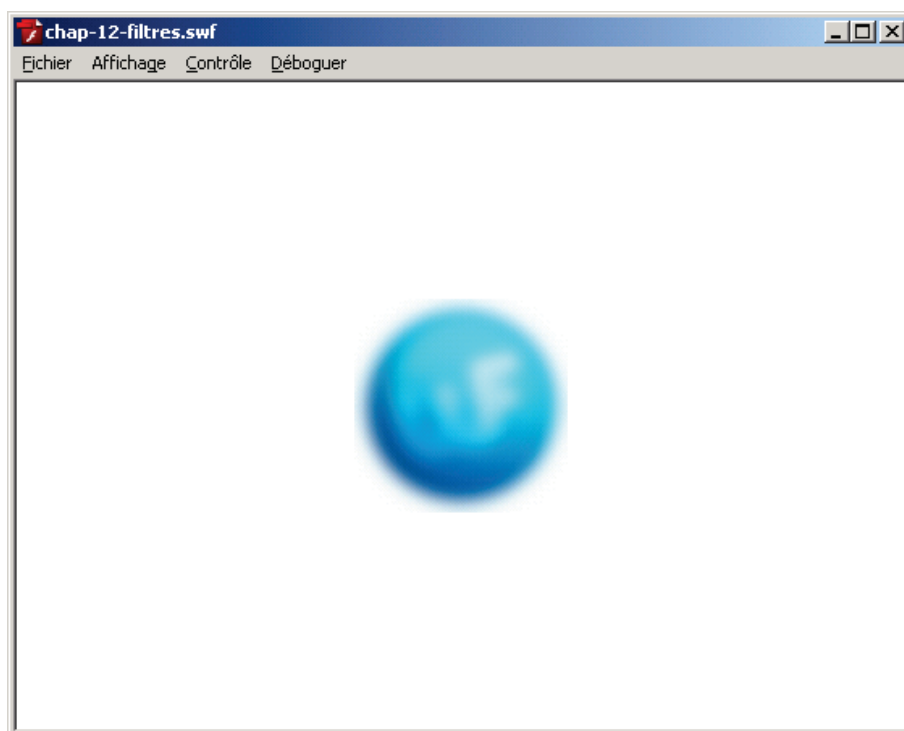
addChild ( monImage );

// création du filtre de flou
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );

// définition d'une surface
var zone:Rectangle = donneesBitmap.rect;

// affectation du filtre de flou sur une surface limitée
donneesBitmap.applyFilter( donneesBitmap, zone, new Point(0,0), filtreFlou );
```

Ainsi, le filtre est appliqué sur la totalité de l'image :



*La figure 12-33. Image entièrement filtrée.*

Lorsqu'un filtre est appliqué à une image bitmap à l'aide de la méthode `applyFilter`, le lecteur ne crée aucun bitmap supplémentaire en mémoire afin de produire le rendu filtré. Les pixels de l'image bitmap sont directement travaillés sur l'instance de `BitmapData`.

Lorsqu'un filtre est associé à un `DisplayObject`, il est possible de faire marche arrière et de supprimer le filtre. A l'inverse, lorsqu'un filtre est appliqué à une image bitmap, les pixels sont définitivement modifiés. Il est impossible de faire marche arrière.

## A retenir

- L'utilisation de filtres sur une instance de `BitmapData` ne crée aucune image bitmap supplémentaire en mémoire.

## Animer un filtre

Afin de donner du mouvement à un filtre nous devons modifier les valeurs correspondantes puis appliquer constamment le filtre afin de mettre à jour l'affichage. Dans le cas de filtres appliqués à un `DisplayObject` nous pouvons écrire le code suivant :

```
stage.addEventListener ( Event.ENTER_FRAME, animFiltre );  
  
var tableauFiltres:Array = new Array();
```

```
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );
tableauFiltres.push ( filtreFlou );
var i:Number = 0;
function animFiltre ( pEvt:Event ):void
{
    // valeur oscillant entre 1 et 21
    var oscillation:Number = Math.floor ( Math.sin ( i += .2 ) * 10 + 11 );

    filtreFlou.blurX = filtreFlou.blurY = oscillation

    pomme.filters = tableauFiltres;
}
```

Le code précédent fait osciller la quantité de flou entre 1 et 21 donnant un effet d'ondulation. Pour reproduire le même effet sur une instance de `BitmapData`, nous utilisons la méthode `applyFilter` :

```
var donneesBitmap:Logo = new Logo ( 0, 0 );
var copieDonneesBitmap:BitmapData = donneesBitmap.clone();
var monImage:Bitmap = new Bitmap ( donneesBitmap );
monImage.x = (stage.stageWidth - monImage.width) / 2;
monImage.y = (stage.stageHeight - monImage.height) / 2
addChild ( monImage );
stage.addEventListener ( Event.ENTER_FRAME, animFiltre );
var tableauFiltres:Array = new Array();
var filtreFlou:BlurFilter = new BlurFilter ( 10, 10, BitmapFilterQuality.HIGH );
tableauFiltres.push ( filtreFlou );
var i:Number = 0;
function animFiltre ( pEvt:Event ):void
{
    // valeur oscillant entre 1 et 20
    var oscillation:Number = Math.floor ( Math.sin ( i += .2 ) * 10 + 11 );

    filtreFlou.blurX = filtreFlou.blurY = oscillation;

    donneesBitmap.copyPixels ( copieDonneesBitmap, copieDonneesBitmap.rect, new
    Point ( 0, 0 ) );
    donneesBitmap.applyFilter ( donneesBitmap, donneesBitmap.rect, new Point (
    0, 0 ), filtreFlou );
}
```

Très souvent, les filtres sont utilisés sur des `DisplayObject` déjà animés. Dans le cas d'une agence Web, le graphiste anime généralement des objets et affecte différents filtres. L'animation du filtre est alors gérée automatiquement par le lecteur Flash.

Pour chaque étape de l'animation, le lecteur Flash met à jour les deux bitmaps en mémoire afin de produire le rendu filtré. Ce processus s'avère complexe et peut ralentir grandement la vitesse d'affichage. Il est donc conseillé de ne pas utiliser un trop grand nombre de filtres sous peine de voir les performances de l'animation chuter.

## Rendu bitmap d'objets vectoriels

La méthode `draw` de la classe `BitmapData` s'avère être une méthode très intéressante. Celle-ci permet de rendre sous forme bitmap n'importe quel `DisplayObject`. Cette fonctionnalité ouvre un grand nombre de possibilités que nous explorerons au cours des prochains chapitres.

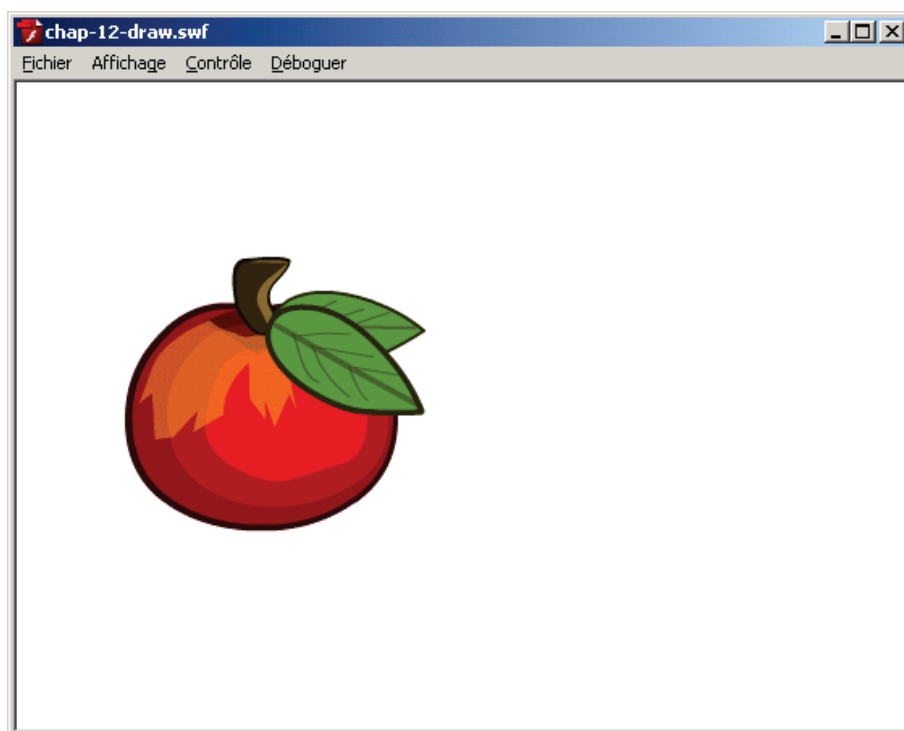
Voici la signature de la méthode `draw` :

```
public function draw(source:IBitmapDrawable, matrix:Matrix = null,
    colorTransform:ColorTransform = null, blendMode:String = null,
    clipRect:Rectangle = null, smoothing:Boolean = false):void
```

Chacun des paramètres permet de travailler sur l'image bitmap :

- `source` : objet source à dessiner.
- `matrix` : matrice de transformation
- `colorTransform` : objet de transformation de couleur permettant de teinter l'image bitmap générée.
- `blendMode` : mode de fondu à appliquer à l'image bitmap générée.
- `clipRect` : surface définissant la zone à dessiner.
- `smoothing` : booléen définissant le lissage de l'image bitmap générée.

Dans un nouveau document Flash CS3, nous plaçons une instance du symbole `Pomme` sur la scène et lui donnons comme nom d'occurrence `pomme`. L'instance a été agrandie d'environ 300% :



*La figure 12-34. Instance du symbole Pomme.*

Nous allons rendre sous forme bitmap une instance du symbole **Pomme** posée sur la scène. Pour cela nous créons une image bitmap afin d'accueillir les pixels dessinés :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCCC );

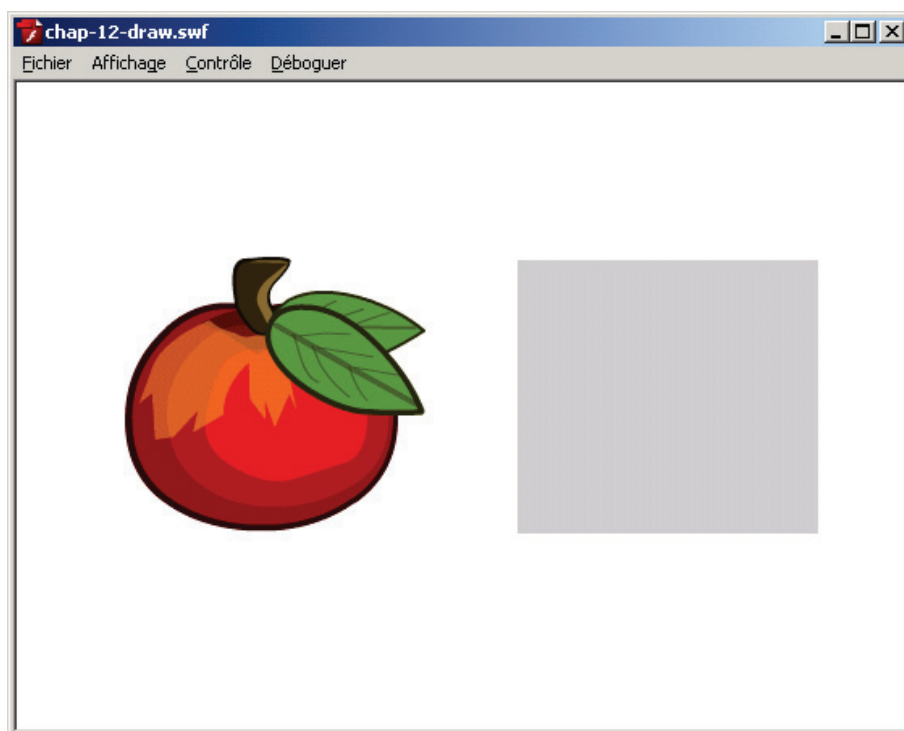
var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

addChild ( monImage );
```

Nous récupérons les dimensions de l'objet à rendre sous forme bitmap afin de créer un bitmap de destination aux dimensions identiques. Souvenez-vous que la création de bitmap entraîne une forte occupation mémoire, chaque pixel doit être optimisé.

La figure 12-35 illustre le résultat :



*La figure 12-35. Image bitmap.*

Puis nous passons à la méthode `draw` l'instance du symbole à rendre sous forme bitmap :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCCC );

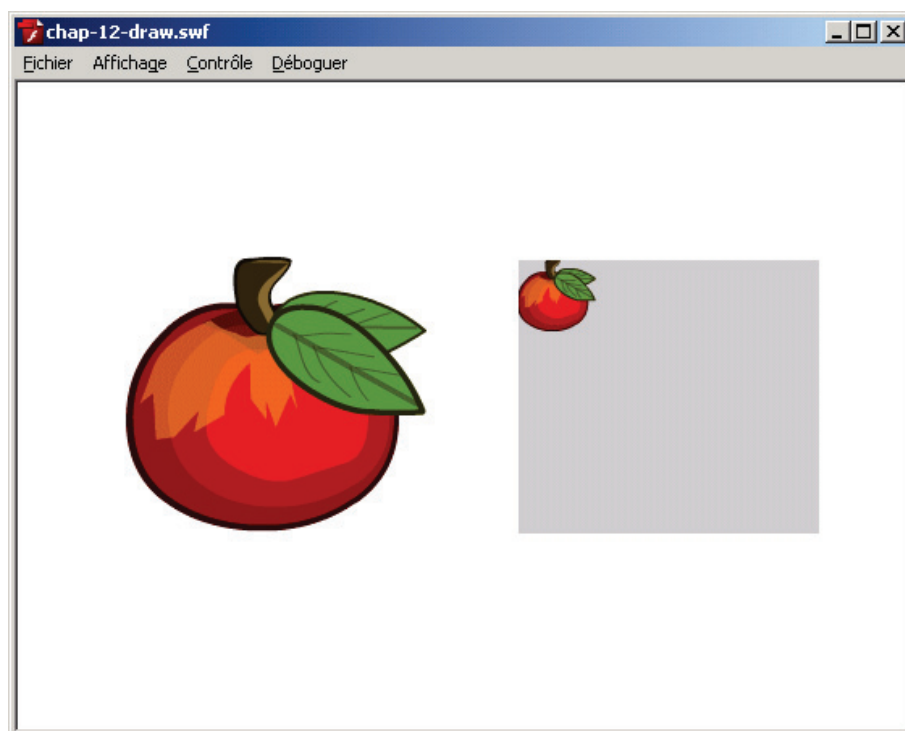
var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

addChild ( monImage );

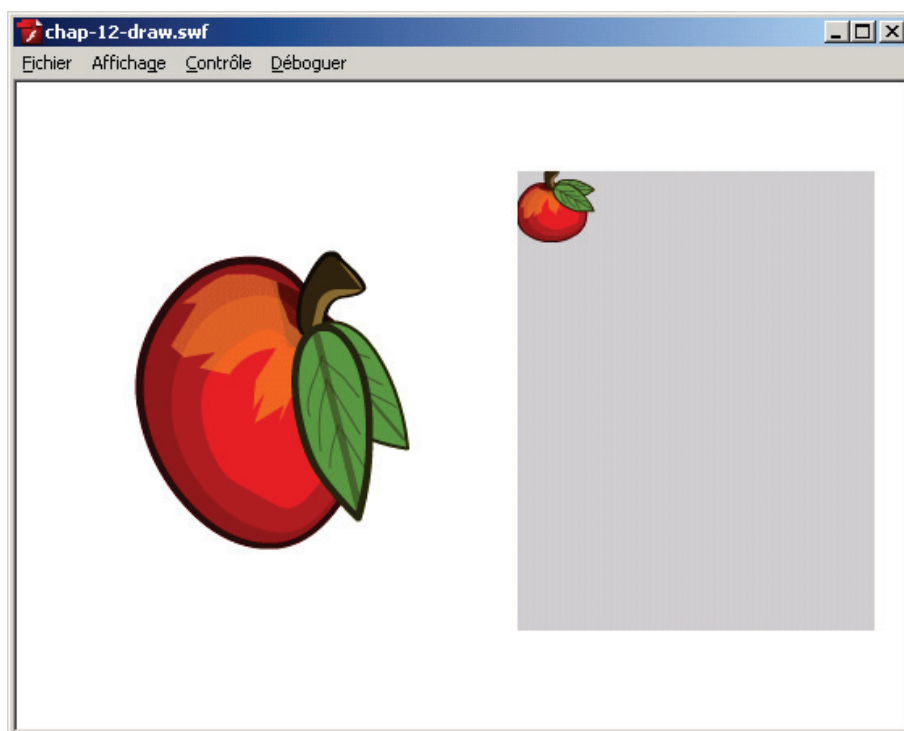
// rend sous forme bitmap les données vectorielles
donneesBitmap.draw ( pomme );
```

Le résultat suivant est généré :



*La figure 12-36. Rendu bitmap.*

Nous remarquons que le rendu bitmap ne possède pas les mêmes dimensions que l'instance. La méthode `draw` dessine par défaut l'objet selon son état en bibliothèque. De la même manière si nous faisons subir une rotation ou un étirement à l'instance du symbole, le rendu bitmap ne reflète pas ces modifications :



*La figure 12-37. Rendu sans transformations.*

Si nous souhaitons prendre en considération les transformations actuelles de l'objet à dessiner, nous devons utiliser une matrice de transformation. Dans notre exemple, nous allons passer la matrice de transformation de l'instance.

Celle-ci est accessible depuis la propriété `matrix` de la classe `Transform`, elle-même accessible depuis la propriété `transform` de tout `DisplayObject` :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

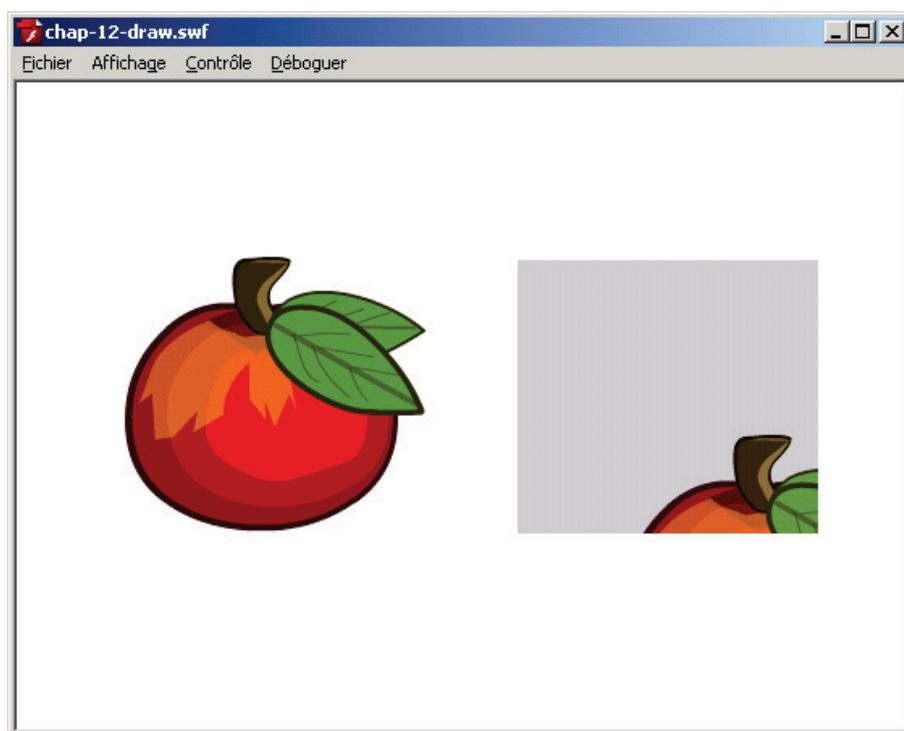
addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// rend sous forme bitmap les données vectorielles en conservant les
transformations
donneesBitmap.draw ( pomme, transformationPomme );
```

Ainsi, le rendu bitmap prend en considération les transformations de l'instance. La figure 12-37 illustre le résultat :





*La figure 12-37. Rendu avec transformations.*

Les pixels ont donc été dessinés avec un décalage correspondant à la position de l'instance par rapport à la scène. Dans notre cas, nous souhaitons conserver les mêmes dimensions.

Afin de décaler les pixels du bitmap, nous utilisons les propriétés `tx` et `ty` de la classe `Matrix`. Celles-ci nous permettent de modifier la translation des pixels dans l'image bitmap générée :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height,
false, 0xCCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y

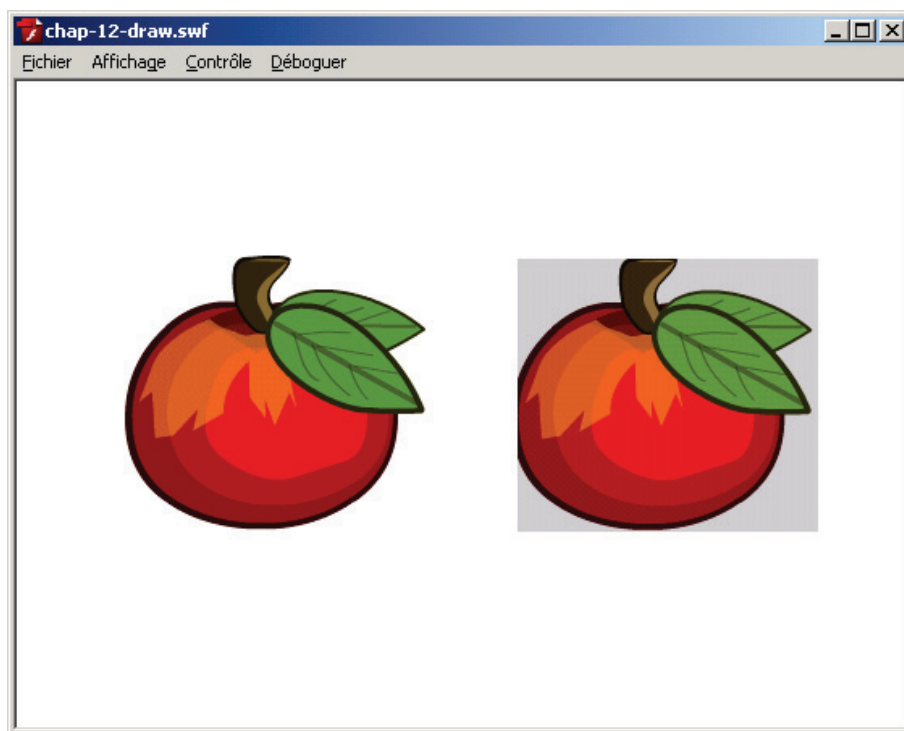
addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

// rend sous forme bitmap les données vectorielles en conservant les
transformations
donneesBitmap.draw ( pomme, transformationPomme );
```

La figure 12-38 illustre le résultat :



*La figure 12-38. Translation.*

Nous pouvons voir que l'image bitmap est tronquée sur le côté gauche et sur le côté supérieur de l'image. Cela vient du fait que les vecteurs composant le symbole `Pomme` passent en dessous de 0 pour l'axe x et y.

---

Attention, lorsque la méthode `draw` est utilisée, le point d'enregistrement de l'objet rasterisé devient le point haut gauche du `BitmapData` généré.

---

Afin d'éviter cette coupure nous pouvons déplacer simplement les pixels en procédant à une simple translation de quelques pixels :

```
// décalage en pixels
var decalage:int = 5;

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width+decalage,
pomme.height+decalage, false, 0xCCCCC );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y - decalage;

addChild ( monImage );
```

```
// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = decalage;
transformationPomme.ty = decalage;

// rend sous forme bitmap les données vectorielles en conservant les
transformations
donneesBitmap.draw ( pomme, transformationPomme );
```

Puis nous adaptons la taille du bitmap de destination et supprimons la couleur de fond :

```
// décalage en pixels
var decalage:int = 5;

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width+decalage,
pomme.height+decalage );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y - decalage;

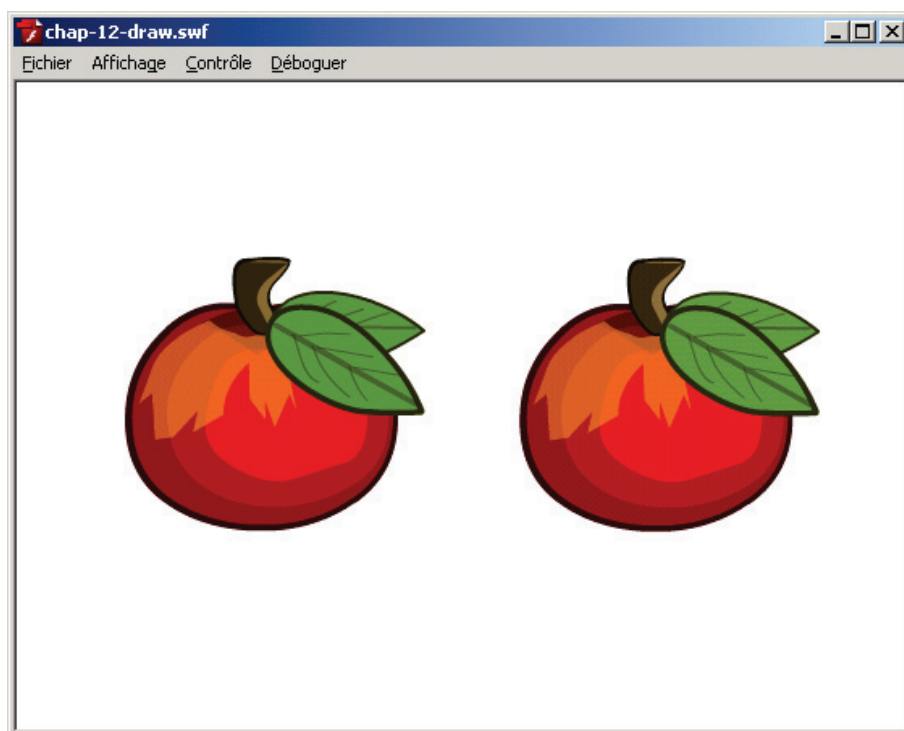
addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = decalage;
transformationPomme.ty = decalage;

// rend sous forme bitmap les données vectorielles en conservant les
transformations
donneesBitmap.draw ( pomme, transformationPomme );
```

La figure 12-39 illustre le résultat :



*La figure 12-39. Rendu bitmap de l'instance.*

Comme vous pouvez l'imaginer, l'évaluation manuelle d'une valeur de translation ne s'avère pas la solution la plus souple. Dans la pratique, il serait idéal de pouvoir déterminer dynamiquement la translation nécessaire à l'image bitmap sans risquer que celle-ci soit coupée lors de la capture.

Afin d'évaluer les débordements de l'objet vectoriel, nous pouvons utiliser la méthode `getBounds` de la classe `DisplayObject`.

Dans le code suivant nous déterminons le débordement du symbole `Pomme` à l'aide de la méthode `getBounds` :

```
var debordement:Rectangle = pomme.getBounds ( pomme );

// affiche : (x=-1, y=-0.5, w=48.75, h=44.5)
trace( debordement );
```

La méthode `getBounds` renvoie un objet `Rectangle` dont les propriétés `x` et `y` nous renvoient les débordements pour les axes respectifs.

Grâce à celles-ci nous pouvons établir une translation dynamique, plus souple qu'un décalage manuel :

```
// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );
```

```

// positionnement de la copie bitmap
monImage.x += 310;
monImage.y += pomme.y;

addChild ( monImage );

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

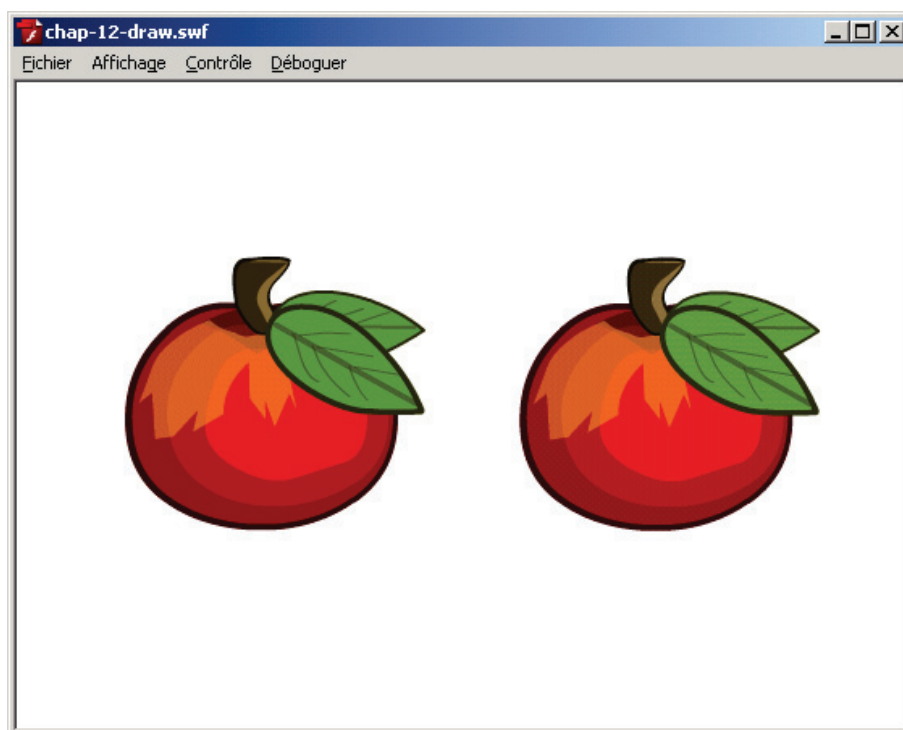
var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les données vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
donneesBitmap.draw ( pomme, transformationPomme );

```

Nous obtenons ainsi de manière dynamique le même résultat que précédemment :



*La figure 12-40. Rendu bitmap de l'instance avec évaluation dynamique du débordement.*

Comme nous l'avons vu précédemment, le symbole `Pomme` possède son point d'enregistrement en haut à gauche. Grâce à la détection du

débordement du symbole, nous pouvons ainsi gérer la capture de symboles ayant leur point d'enregistrement au centre.

Si nous déplaçons le point d'enregistrement du symbole `Pomme` en son centre, la capture fonctionne parfaitement sans aucune modification du code excepté le positionnement de l'image bitmap :

```
// positionnement de la copie bitmap
monImage.x += 310;
monImage.y = pomme.y - (pomme.height * .5);
```

Nous allons à présent récupérer la couleur de chaque pixel composant notre image bitmap. Pour cela nous devons cliquer sur la pomme bitmap et accéder à la couleur du pixel cliqué.

La classe `Bitmap` n'est pas une sous-classe de la classe `InteractiveObject`, ainsi si nous souhaitons interagir avec une image bitmap nous devons au préalable la placer dans un conteneur de type `InteractiveObject`. Pour cela, nous utilisons la classe `flash.display.Sprite`.

Nous plaçons l'image bitmap au sein d'un conteneur :

```
// création d'un conteneur pour l'image bitmap
var conteneurImage:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurImage );

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// ajout de l'image au conteneur
conteneurImage.addChild ( monImage );

// positionnement de la copie bitmap
conteneurImage.x += 310;
conteneurImage.y += pomme.y - (pomme.height * .5);

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les données vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
```

```
| donneesBitmap.draw ( pomme, transformationPomme );
```

Puis nous écoutons différents événements liés à l'interactivité sur le conteneur afin de récupérer la couleur du pixel survolé :

```
// création d'un conteneur pour l'image bitmap
var conteneurImage:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurImage );

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// ajout de l'image au conteneur
conteneurImage.addChild ( monImage );

// positionnement de la copie bitmap
conteneurImage.x += 310;
conteneurImage.y += pomme.y - (pomme.height * .5);

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les données vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
donneesBitmap.draw ( pomme, transformationPomme );

// écoute des événements MouseEvent.MOUSE_DOWN et MouseEvent.MOUSE_UP
// auprès du conteneur et de l'objet Stage (cela permet de capter le
relâchement en dehors du conteneur)
conteneurImage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function clicSouris ( pEvt:MouseEvent ):void
{
    bougeSouris ( pEvt );

    pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

function relacheSouris ( pEvt:MouseEvent ):void
{
    conteneurImage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}
```

```
}

function bougeSouris ( pEvt:MouseEvent ):void

{

    // accède à la couleur du pixel survolé
    var couleurPixel:Number = donneesBitmap.getPixel32 ( pEvt.localX,
pEvt.localY );

    // affiche : ffd8631f
    trace( couleurPixel.toString ( 16 ) );

}
```

Enfin, nous ajoutons une image bitmap de taille réduite et une légende afin d’afficher la couleur survolée :

```
// création d'un conteneur pour l'image bitmap
var conteneurImage:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( conteneurImage );

// création des données bitmaps aux dimensions de l'image source
var donneesBitmap:BitmapData = new BitmapData ( pomme.width, pomme.height );

var monImage:Bitmap = new Bitmap ( donneesBitmap );

// ajout de l'image au conteneur
conteneurImage.addChild ( monImage );

// positionnement de la copie bitmap
conteneurImage.x += 310;
conteneurImage.y += pomme.y - (pomme.height * .5);

// transformations actuelles de la pomme
var transformationPomme:Matrix = pomme.transform.matrix;

// repositionne les pixels
transformationPomme.tx = 0;
transformationPomme.ty = 0;

var debordement:Rectangle = pomme.getBounds ( pomme );

// décalage répercuté sur la matrice de transformation
transformationPomme.translate ( -debordement.x * pomme.scaleX, -debordement.y *
pomme.scaleY );

// rend sous forme bitmap les données vectorielles en conservant les
transformations
// grâce à la translation dynamique réalisée, l'image bitmap n'est pas coupée
donneesBitmap.draw ( pomme, transformationPomme );

// écoute des événements MouseEvent.MOUSE_DOWN et MouseEvent.MOUSE_UP
// auprès du conteneur et de l'objet Stage (cela permet de capter le
relâchement en dehors du conteneur)
conteneurImage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

function clicSouris ( pEvt:MouseEvent ):void
```



```
{
    bougeSouris ( pEvt );

    pEvt.currentTarget.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

function relacheSouris ( pEvt:MouseEvent ):void
{
    conteneurImage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
}

function bougeSouris ( pEvt:MouseEvent ):void
{
    // accède à la couleur du pixel survolé
    var couleurPixel:Number = donneesBitmap.getPixel32 ( pEvt.localX,
pEvt.localY );

    // force le rafraichissement
    pEvt.updateAfterEvent();

    // remplissage du bitmap d'aperçu
    apercuCouleur.fillRect ( apercuCouleur.rect, couleurPixel );
}

// création d'une image bitmap d'aperçu de selection de couleur
var apercuCouleur:BitmapData = new BitmapData ( 120, 60, false, 0 );

var imageApercu:Bitmap = new Bitmap ( apercuCouleur );

// positionnement
imageApercu.x = 210;
imageApercu.y = 320;

// ajout à la liste d'affichage
addChild ( imageApercu );

// création d'une légende
var legende:TextField = new TextField();

legende.x = 210;
legende.y = 300;

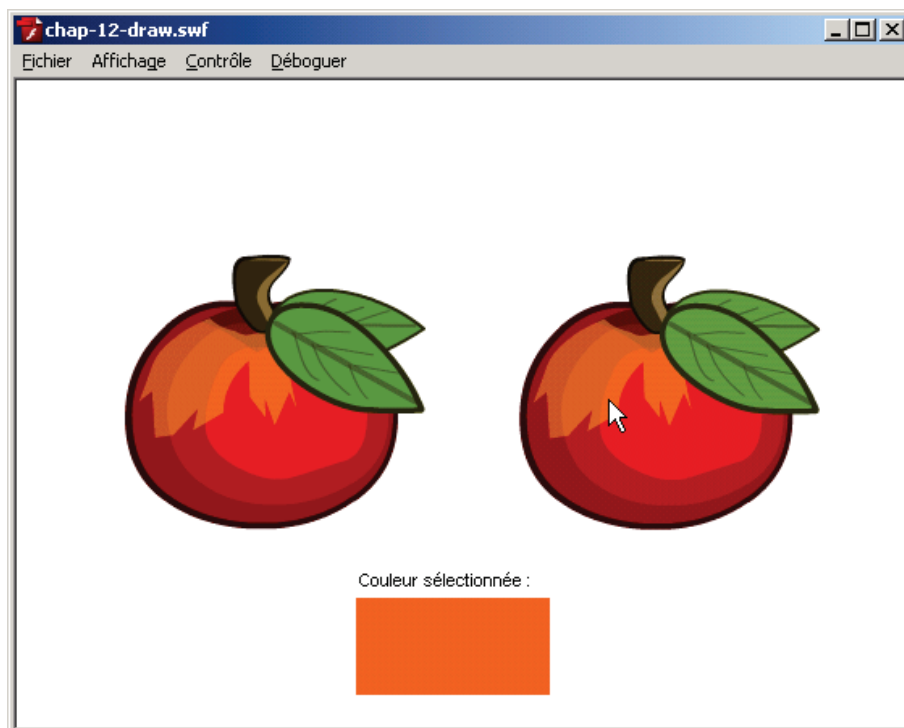
// redimensionnement automatique
legende.autoSize = TextFieldAutoSize.LEFT;

// formatage du texte
var formatage:TextFormat = new TextFormat();
formatage.font = "Arial";
formatage.size = 11;

// affectation du formatage
legende.defaultTextFormat = formatage
```

```
legende.text = "Couleur sélectionnée :";  
addChild ( legende );
```

A la sélection d'une zone sur l'image bitmap nous obtenons un aperçu comme l'illustre la figure 12-41 :



*La figure 12-41. Sélection d'une couleur.*

La méthode `getPixel32` nous permet de récupérer la couleur de chaque pixel survolé. Puis nous colorons l'image bitmap `aperçuCouleur` afin d'obtenir un aperçu.

La capture d'éléments vectoriels sous forme bitmap offre de nouvelles possibilités comme la compression d'images au format JPEG ou l'encodage au format PNG, GIF ou autres.

Rendez-vous au chapitre 20 intitulé `ByteArray` pour plus de détails concernant l'encodage d'images sous différents formats.

## A retenir

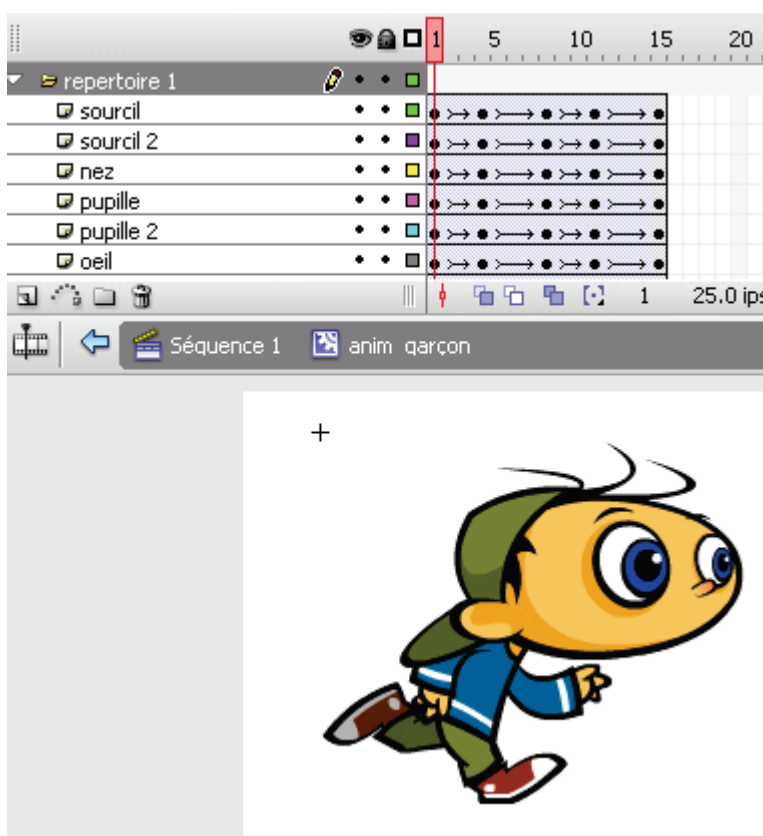
- La méthode `draw` permet de rendre sous forme bitmap un élément vectoriel.
- Afin d'appliquer des transformations au sein du bitmap rendu, nous utilisons une matrice de transformation.

## Optimiser les performances

Nous allons utiliser la classe `BitmapData` afin d'optimiser les performances d'une animation traditionnelle.

En créant une classe `AnimationBitmap` nous allons capturer chaque état de l'animation vectorielle sous forme d'image bitmap, puis simuler l'animation en les affichant successivement. Cela va nous permettre d'améliorer les performances de rendu et réduire l'occupation processeur.

Dans un nouveau document Flash CS3 nous utilisons un personnage animé, la figure 12-42 illustre l'animation :



*La figure 12-42. Animation du personnage.*

Par le panneau *Liaison* nous lions le symbole animé à une classe `Garcon` puis nous l'instancions avec le code suivant :

```
// instancie l'animation
var monPersonnage:Garcon = new Garcon();
```

A coté du document flash en cours nous créons un répertoire bitmap puis nous plaçons à l'intérieur une classe nommée **AnimationBitmap**.

Celle-ci contient le code suivant :

```
package bitmap

{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.MovieClip;
    import flash.events.TimerEvent;
    import flash.geom.Matrix;
    import flash.utils.Timer;

    public class AnimationBitmap extends Bitmap
    {

        // tableau stockant chaque étape de l'animation sous forme bitmap
        private var tableauBitmaps:Array;
        // sauve une référence vers l'objet vectoriel à animer
        private var animationCible:MovieClip;
        // création d'un minuteur pour gérer l'animation
        private var minuteur:Timer;
        // variable d'incrémentation
        private var i:int;

        public function AnimationBitmap ( pCible:MovieClip, pVitesse:int=100 )
        {

            i = 0;
            animationCible = pCible;
            tableauBitmaps = new Array();
            minuteur = new Timer ( pVitesse, 0 );
            minuteur.addEventListener ( TimerEvent.TIMER, anime );
            genereEtapes ();

        }

        private function genereEtapes ():void
        {

            // récupère le nombre d'image totale
            var lng:int = animationCible.totalFrames;
            var etapeAnimation:BitmapData;

            for ( var i:int = 0; i< lng; i++ )

            {

                // parcourt l'animation
                animationCible.gotoAndStop(i);
                // crée un bitmap pour chaque étape
```

```
        etapeAnimation = new BitmapData ( animationCible.width + 30,
animationCible.height, true, 0 );
        // rend l'image sous forme bitmap
        etapeAnimation.draw ( animationCible );
        tableauBitmaps.push ( etapeAnimation );

    }

    minuteur.start();

}

private function anime ( pEvt:TimerEvent ):void
{
    // met à jour l'affichage, l'animation se joue en boucle
    bitmapData = tableauBitmaps[i++%tableauBitmaps.length];
}

}

}
```

Puis nous passons au constructeur de la classe `AnimationBitmap` l'objet vectoriel à animer sous forme bitmap puis sa vitesse de lecture :

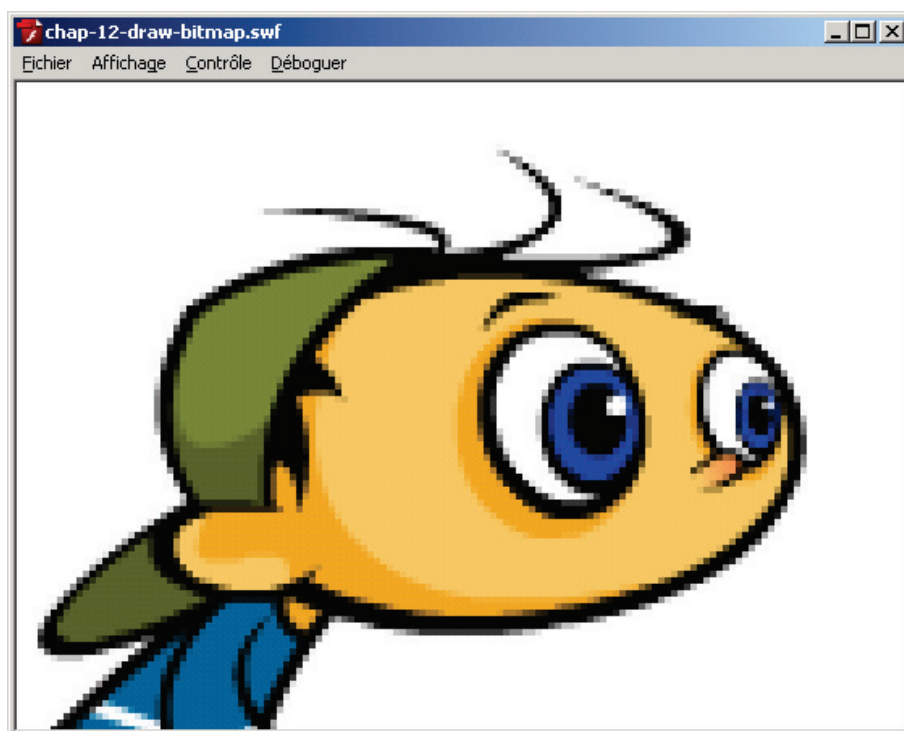
```
// instancie l'animation
var monPersonnage:Garcon = new Garcon();

// crée l'animation
var animPersonnage:AnimationBitmap = new AnimationBitmap ( monPersonnage, 30 );

// positionnement
animPersonnage.x = 20;
animPersonnage.y = 100;

// ajout à la liste d'affichage
addChild ( animPersonnage );
```

Si nous zoomons sur l'animation générée, nous pouvons apercevoir qu'il s'agit d'une succession d'image bitmaps, le rendu est pixelisé :



*La figure 12-43. Animation bitmap.*

Il est possible de choisir la vitesse de chaque animation. Dans le code suivant nous créons trois personnages. Chacun d’eux possède sa propre vitesse :

```
// instancie l'animation
var monPersonnage:Garcon = new Garcon();

// crée l'animation
var animPersonnage:AnimationBitmap = new AnimationBitmap ( monPersonnage, 30 );

// positionnement
animPersonnage.x = 20;
animPersonnage.y = 100;

// ajout à la liste d'affichage
addChild ( animPersonnage );

var deuxiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 90 );

deuxiemeAnimPersonnage.x = 190;
deuxiemeAnimPersonnage.y += 100;

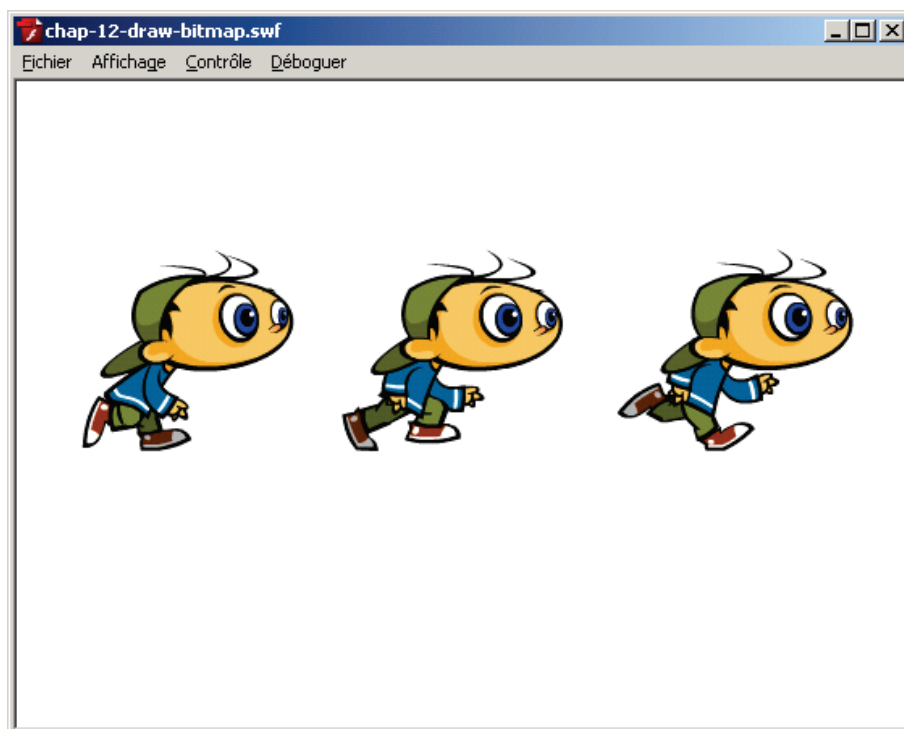
addChild ( deuxiemeAnimPersonnage );

var troisiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 150 );

troisiemeAnimPersonnage.x = 370;
troisiemeAnimPersonnage.y = 100;

addChild ( troisiemeAnimPersonnage );
```

La figure 12-44 illustre les trois animations produites :



*La figure 12-43. Animation bitmap.*

En utilisant des images bitmaps pour simuler l'animation nous avons amélioré la fluidité de l'animation et réduit l'occupation processeur.

Une dernière astuce va nous permettre d'optimiser la vitesse de rendu des animations. Lorsque nous générons les différentes étapes de l'animation sous forme bitmap nous passons la scène en qualité optimale. Puis, une fois les images bitmaps rendues nous passons la scène en qualité réduite.

Les bitmaps affichés restent ainsi lissées tout en ayant passé l'animation en qualité réduite :

```
// passe la qualité du rendu en optimale
stage.quality = StageQuality.HIGH;

// instancie l'animation
var monPersonnage:Garcon = new Garcon();

// crée l'animation
var animPersonnage:AnimationBitmap = new AnimationBitmap ( monPersonnage, 30 );

// positionnement
animPersonnage.x = 20;
animPersonnage.y = 100;

// ajout à la liste d'affichage
addChild ( animPersonnage );
```

```
var deuxiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 90 );

deuxiemeAnimPersonnage.x = 190;
deuxiemeAnimPersonnage.y += 100;

addChild ( deuxiemeAnimPersonnage );

var troisiemeAnimPersonnage:AnimationBitmap = new AnimationBitmap (
monPersonnage, 150 );

troisiemeAnimPersonnage.x = 370;
troisiemeAnimPersonnage.y = 100;

addChild ( troisiemeAnimPersonnage );

// une fois l'animation bitmap générée, nous repassons en qualité réduite
stage.quality = StageQuality.LOW;
```

Nous bénéficions ainsi des performances d'une animation lue en qualité réduite sans dégrader la qualité de l'animation. Nous verrons lors du chapitre 16 intitulé *Gestion du texte* d'autres techniques d'optimisations du rendu.

Peut être avez-vous pensé activer la mise en cache des bitmaps sur chaque personnage afin d'optimiser le rendu ?

Souvenez-vous, lorsque la tête de lecture d'un objet graphique est en mouvement et que la mise en cache des bitmaps est activée, pour chaque nouvelle image le lecteur met à jour le bitmap créé en mémoire et ralentit les performances de l'animation.

L'intérêt majeur de la classe `AnimationBitmap` repose sur la création d'images bitmaps indépendantes qui ne sont pas mises à jour quelles que soient les transformations appliquées à l'animation générée.

## A retenir

- L'utilisation de filtres sur une instance de `BitmapData` ne crée aucune image bitmap supplémentaire en mémoire.

Au cours du prochain chapitre nous découvrirons comment charger des données non graphiques dans nos applications.



# 13

## Chargement du contenu

<b>LA CLASSE LOADER.....</b>	<b>1</b>
CHARGER UN ÉLÉMENT EXTERNE.....	2
<b>LA CLASSE LOADERINFO.....</b>	<b>5</b>
<b>INTERAGIR AVEC LE CONTENU .....</b>	<b>18</b>
<b>CREER UNE GALERIE .....</b>	<b>26</b>
LA COMPOSITION .....	29
REDIMENSIONNEMENT AUTOMATIQUE.....	34
GESTION DU LISSAGE .....	44
<b>PRECHARGER LE CONTENU .....</b>	<b>48</b>
<b>INTERROMPRE LE CHARGEMENT.....</b>	<b>57</b>
<b>COMMUNIQUER ENTRE DEUX ANIMATIONS .....</b>	<b>58</b>
<b>MODELE DE SECURITE DU LECTEUR FLASH.....</b>	<b>61</b>
PROGRAMMATION CROISÉE.....	62
UTILISER UN FICHIER DE RÉGULATION .....	63
<b>CONTEXTE DE CHARGEMENT .....</b>	<b>65</b>
<b>CONTOURNER LES RESTRICTIONS DE SECURITE.....</b>	<b>66</b>
<b>BIBLIOTHEQUE PARTAGEE .....</b>	<b>72</b>
<b>DESACTIVER UNE ANIMATION CHARGEE .....</b>	<b>79</b>
<b>COMMUNICATION AVM1 ET AVM2.....</b>	<b>81</b>

### La classe Loader

ActionScript 3 élimine les nombreuses fonctions et méthodes existantes dans les précédentes versions d'ActionScript telles `loadMovie` ou `loadMovieNum` et étend le concept de la classe `MovieClipLoader` introduite avec Flash MX 2004.

Afin de charger différents types de contenus tels des images ou des animations nous utilisons la classe `flash.display.Loader`. Celle-ci hérite de la classe `DisplayObjectContainer` et peut donc être ajoutée à la liste d’affichage.

D’autres classes telles `flash.net.URLLoader` ou `flash.net.URLStream` peuvent être utilisées afin de charger tout type de contenu, mais leur utilisation s’avère plus complexe. Nous reviendrons en détail sur ces classes au cours du chapitre 14 intitulé *Charger et envoyer des données*.

Nous allons commencer notre apprentissage en chargeant différents types de contenu, puis nous continuerons notre exploration de la classe `Loader` en créant une galerie photo.

## A retenir

- La classe `Loader` permet le chargement de contenu graphique seulement.
- Celle-ci hérite de la classe `DisplayObjectContainer` et peut donc être ajoutée à la liste d’affichage.
- Il faut considérer la classe `Loader` comme un objet d’affichage.

## Charger un élément externe

La classe `Loader` permet de charger des images au format PNG, JPEG, JPEG progressif et GIF non animé. Des animations SWF peuvent aussi être chargées, nous verrons comment communiquer avec celles-ci en fin de chapitre.

Afin de charger un de ces fichiers, nousinstancions tout d’abord la classe `Loader` :

```
| var chargeur:Loader = new Loader();
```

Puis nous appelons la méthode `load` dont voici la signature :

```
| public function load(request:URLRequest, context:LoaderContext = null):void
```

Celle-ci requiert deux paramètres dont voici le détail :

- `request` : un objet `URLRequest` contenant l’adresse de l’élément à charger.
- `context` : un objet `LoaderContext` définissant le contexte de l’élément chargé.

La classe `URLRequest` est utilisée pour toute connexion externe HTTP liée aux classes `URLLoader`, `URLStream`, `Loader` et `FileReference`.

Dans le code suivant, nous enveloppons l'URL à atteindre dans un objet `URLRequest` :

```
// création du chargeur
var chargeur:Loader = new Loader();

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load ( maRequete );
```

L'utilisation d'un objet `URLRequest` diffère des précédentes versions d'ActionScript où une simple chaîne de caractères était passée aux différentes fonctions liées au chargement externe. Certaines classes ActionScript 3 telles `Socket`, `URLStream`, `NetConnection` et `NetStream` font néanmoins exception et ne nécessitent pas d'objet `URLRequest`.

Nous reviendrons en détail sur la classe `URLRequest` au cours du chapitre 14 intitulé *Charger et envoyer des données*.

Nous ne nous intéressons pas pour le moment au paramètre `context` qui n'est pas utilisé par défaut. Nous verrons plus loin dans ce chapitre l'intérêt de ce dernier.

Afin de rendre graphiquement le contenu chargé nous devons ajouter l'objet `Loader` à la liste d'affichage :

```
// création du chargeur
var chargeur:Loader = new Loader();

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

// ajout à la liste d'affichage
addChild ( chargeur );
```

L'image est alors affichée :



*Figure 13-1. Image chargée dynamiquement.*

Il est important de signaler qu’une instance de la classe `Loader` ne peut contenir qu’un seul enfant. Ainsi, lorsque la méthode `load` est exécutée, le contenu précédent est automatiquement remplacé.

Ainsi, les images bitmap sont automatiquement supprimées de la mémoire, lorsque celles-ci sont déchargées. Ce n’est pas le cas des animations dont nous devons gérer la désactivation complète. Nous reviendrons sur cette contrainte en fin de chapitre.

Comme depuis toujours, le chargement des données externes en ActionScript est asynchrone, cela signifie que l’instruction `load` n’est pas bloquante. Le code continue d’être exécuté une fois le chargement démarré. Nous serons avertis de la fin du chargement des données plus tard dans le temps grâce à un événement approprié.

---

## A retenir

---

- La méthode `load` permet de charger le contenu externe.
- Afin de voir le contenu chargé, l'objet `Loader` doit être ajouté à la liste d'affichage.

## La classe `LoaderInfo`

Afin de pouvoir accéder au contenu chargé, ou d'être averti des différentes phases du chargement nous utilisons les événements diffusés par la classe `flash.display.LoaderInfo`. Celle-ci contient des informations relatives au contenu chargé de type bitmap ou SWF.

Il existe deux moyens d'accéder à un objet `LoaderInfo` :

- Par la propriété `contentLoaderInfo` de l'objet `Loader`.
- Par la propriété `loaderInfo` de n'importe quel `DisplayObject`.

Lorsque nous accédons à la propriété `contentLoaderInfo` de la classe `Loader` nous récupérons une référence à l'objet `LoaderInfo` gérant le chargement de l'élément.

Si nous tentons d'écouter les différents événements liés au chargement directement auprès de l'objet `Loader`, aucun événement n'est diffusé :

```
// création du chargeur
var chargeur:Loader = new Loader();

// tentative d'écoute de l'événement Event.COMPLETE
chargeur.addEventListener ( Event.COMPLETE, contenuCharge );

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

// ajout à la liste d'affichage
addChild ( chargeur );

function contenuCharge ( pEvt:Event ):void
{
    trace( pEvt );
}
```

Ainsi, l'écoute des différents événements propres au chargement, se fait auprès de l'objet `LoaderInfo` accessible par la propriété `contentLoaderInfo` de l'objet `Loader` :

```
// création du chargeur
```

```
var chargeur:Loader = new Loader();

// écoute de l'événement Event.COMPLETE
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, contenuCharge
);

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

// ajout à la liste d'affichage
addChild ( chargeur );

function contenuCharge ( pEvt:Event ):void
{
    // affiche : [Event type="complete" bubbles=false cancelable=false
    eventPhase=2]
    trace( pEvt );

    // affiche : [object LoaderInfo]
    trace( pEvt.target );
}
```

De nombreux événements sont diffusés par la classe `LoaderInfo`, voici le détail de chacun d'entre eux :

- `Event.OPEN` : diffusé lorsque le lecteur commence à charger le contenu.
- `ProgressEvent.PROGRESS` : diffusé lorsque le chargement est en cours. Celui-ci renseigne sur le nombre d'octets chargés et totaux. Sa fréquence de diffusion est liée à la vitesse de téléchargement de l'élément.
- `Event.INIT` : diffusé lorsqu'il est possible d'accéder au code d'un SWF chargé. Cet événement n'est pas utilisé dans le cas de chargement d'image bitmap.
- `Event.COMPLETE` : diffusé lorsque le chargement est terminé.
- `Event.UNLOAD` : diffusé lorsque la méthode `unload` est appelée ou qu'un nouvel élément va être chargé pour remplacer le précédent.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque le chargement échoue.
- `HTTPStatusEvent.HTTP_STATUS` : indique le code d'état de la requête HTTP.

Afin de gérer avec finesse le chargement nous pouvons écouter chacun des événements :

```
// création du chargeur
var chargeur:Loader = new Loader();

// référence à l'objet LoaderInfo
var cli:LoaderInfo = chargeur.contentLoaderInfo;
```

```
// écoute des événements liés au chargement
cli.addEventListener ( Event.OPEN, debutChargement );
cli.addEventListener ( Event.INIT, initialisation );
cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
cli.addEventListener ( Event.COMPLETE, chargementTermine );
cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );
cli.addEventListener ( HTTPStatusEvent.HTTP_STATUS, echecHTTP );
cli.addEventListener ( Event.UNLOAD, suppressionContenu );

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

// ajout à la liste d'affichage
addChild ( chargeur );

function debutChargement ( pEvt:Event ):void
{
    // affiche : [Event type="open" bubbles=false cancelable=false
    eventPhase=2]
    trace( pEvt );
}

function initialisation ( pEvt:Event ):void
{
    // affiche : [Event type="init" bubbles=false cancelable=false
    eventPhase=2]
    trace( pEvt );
}

function chargement ( pEvt:ProgressEvent ):void
{
    // affiche : [ProgressEvent type="progress" bubbles=false cancelable=false
    eventPhase=2 bytesLoaded=0 bytesTotal=5696]
    trace( pEvt );
}

function chargementTermine ( pEvt:Event ):void
{
    // affiche : [Event type="complete" bubbles=false cancelable=false
    eventPhase=2]
    trace( pEvt );
}

function echecChargement ( pEvt:IOErrorEvent ):void
{

```

```
        trace( pEvt );
    }

    function echecHTTP ( pEvt:HTTPStatusEvent ):void
    {
        trace( pEvt );
    }

    function suppressionContenu ( pEvt:Event ):void
    {
        // affiche : [Event type="unload" bubbles=false cancelable=false
        // eventPhase=2]
        trace( pEvt );
    }
}
```

Si nous tentons d'accéder au contenu chargé avant l'événement `Event.COMPLETE` l'accès à la propriété `content` lève une erreur à l'exécution. Dans le code suivant nous tentons d'accéder au contenu durant l'événement `ProgressEvent.PROGRESS` :

```
function chargement ( pEvt:ProgressEvent ):void
{
    trace( pEvt.target.content );
}
```

L'erreur suivante est levée à l'exécution :

```
Error: Error #2099: L'objet en cours de chargement n'est pas suffisamment
chargé pour fournir ces informations.
```

Une fois le contenu totalement chargé, nous y accédons grâce à la propriété `content` de la classe `LoaderInfo` :

```
function chargementTermine ( pEvt:Event ):void
{
    // affiche : [object Bitmap]
    trace( pEvt.target.content );
}
```

Mais aussi par la propriété `content` de l'objet `Loader` :

```
// affiche : [object Bitmap]
trace( chargeur.content );
```

A l'aide des différentes propriétés de la classe `LoaderInfo` nous obtenons différents types d'informations :



```
function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // accès aux informations liées au contenu chargé
    var contenu:Bitmap = Bitmap ( objetLoaderInfo.content );

    var largeur:Number = objetLoaderInfo.width;
    var hauteur:Number = objetLoaderInfo.height;

    var urlContenu:String = objetLoaderInfo.url;

    var type:String = objetLoaderInfo.contentType;

    // affiche : [object Bitmap]
    trace ( contenu );

    // affiche : 167 65
    trace ( largeur, hauteur );

    // affiche :
    file:///K:/Work/Bytearray.org/O%27Reilly/Pratique%20d%27ActionScript/Chapitre
    %2013/flas/photo.jpg
    trace ( urlContenu );

    // affiche : image/jpeg
    trace ( type );
}
```

Lors du chapitre 12 intitulé *Programmation Bitmap*, nous avons vu que toutes les données bitmap étaient assimilées à des objets `BitmapData`. Dans le cas d'une image chargée dynamiquement, les données bitmap sont enveloppées au préalable par un objet `flash.display.Bitmap` afin de pouvoir être rendues.

Dans le code suivant, nous récupérerons les dimensions de l'image chargée ainsi que les données bitmap la composant :

```
function chargementTermine ( pEvt:Event ):void
{
    // référence le contenu
    var contenu:DisplayObject = pEvt.target.content;

    // si le contenu chargé est une image
    if ( contenu is Bitmap )
    {
        // transtype en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        // affiche : 167 65
        trace( image.width, image.height );

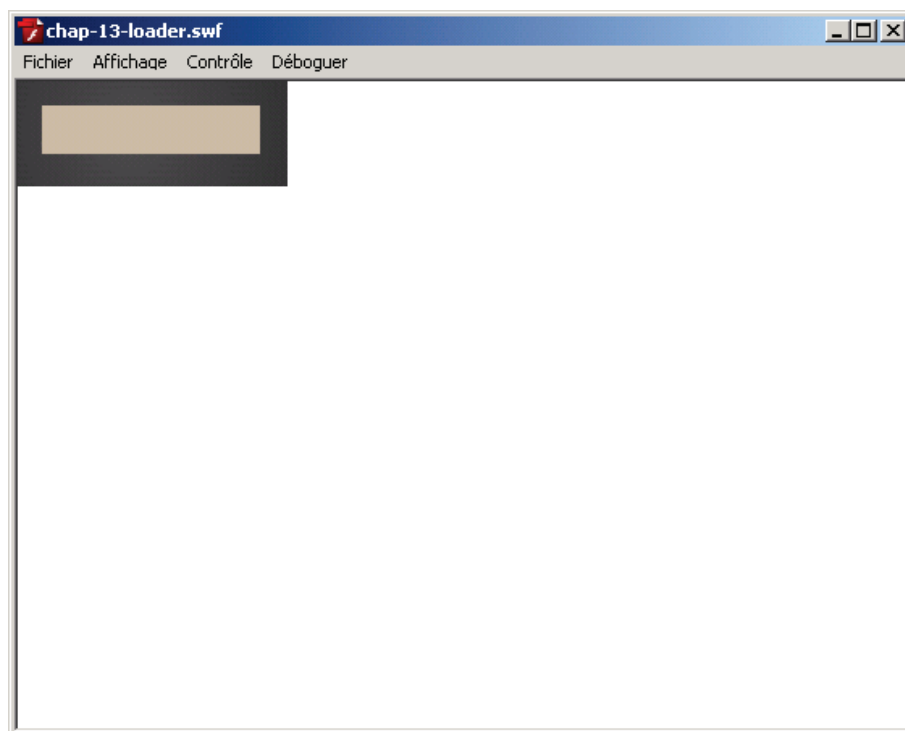
        //affiche : [object BitmapData]
```

```
        trace( image.bitmapData );  
    }  
}
```

Nous pouvons effacer une partie de l'image chargée, en peignant directement dessus à l'aide de la méthode `fillRect` de la classe `BitmapData` :

```
function chargementTermine ( pEvt:Event ):void  
{  
    // référence le contenu  
    var contenu:DisplayObject = pEvt.target.content;  
  
    // si le contenu chargé est une image  
    if ( contenu is Bitmap )  
    {  
        // transtype en tant qu'image  
        var image:Bitmap = Bitmap ( contenu );  
  
        var donneesBitmap:BitmapData = image.bitmapData;  
  
        donneesBitmap.fillRect( new Rectangle ( 15, 15, 135, 30 ), 0xC8BCA4 );  
    }  
}
```

La figure 13-2 illustre le résultat :



*Figure 13-2. Modification des données bitmap.*

La partie centrale de l'image chargée est peinte de pixels beiges.

Souvenez-vous que les événements sont diffusés depuis l'objet `LoaderInfo`. Ainsi la propriété `target` ne référence pas l'objet `Loader` mais l'objet `LoaderInfo` associé à l'élément chargé.

Afin d'accéder à l'objet `Loader` associé à l'objet `LoaderInfo`, nous utilisons sa propriété `loader` :

```
function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // affiche : true
    trace( objetLoaderInfo.loader == chargeurImage );
}
```

De la même manière, la propriété `contentLoaderInfo` de l'objet `Loader` pointe vers le même objet `LoaderInfo` que celui référencé par la propriété `loaderInfo` de l'élément chargé :

```
function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    var objetLoader:Loader = pEvt.target.loader;

    // affiche : true
    trace( objetLoader.contentLoaderInfo == objetLoaderInfo.content.loaderInfo );
}
```

L'objet `LoaderInfo` est donc partagé entre deux environnements :

L'animation chargeant le contenu et le contenu lui-même.

Il est important de noter que bien qu'il s'agisse d'une sous-classe de `DisplayObjectContainer`, la classe `Loader` ne possède pas toutes les capacités propres à ce type.

Certaines propriétés telles `numChildren` peuvent être utilisées :

```
function chargementTermine ( pEvt:Event ):void
{
    var objetLoader:Loader = pEvt.target.loader;

    // affiche : true
```

```
    trace( objetLoader.numChildren );  
}
```

Il est en revanche impossible d'appeler les différentes méthodes de manipulation d'objets enfants sur celle-ci. Dans le code suivant nous tentons d'appeler la méthode `removeChildAt` sur l'objet `chargeurImage` :

```
function chargementTermine ( pEvt:Event ):void  
{  
    var objetLoader:Loader = pEvt.target.loader;  
    // tentative de suppression du premier enfant de l'objet Loader  
    objetLoader.removeChildAt(0);  
}
```

L'erreur à l'exécution suivante est levée :

```
// affiche : Error: Error #2069: La classe Loader ne met pas en oeuvre cette  
méthode.
```

Afin de supprimer l'image chargée, nous devons appeler la méthode `unload` de la classe `Loader`.

Dans le code suivant, nous supprimons l'image chargée aussitôt le chargement terminé :

```
function chargementTermine ( pEvt:Event ):void  
{  
    // référence le contenu  
    var contenu:DisplayObject = pEvt.target.content;  
    // si le contenu chargé est une image  
    if ( contenu is Bitmap )  
    {  
        // transtypage en tant qu'image  
        var image:Bitmap = Bitmap ( contenu );  
        var donneesBitmap:BitmapData = image.bitmapData;  
        donneesBitmap.fillRect( new Rectangle ( 15, 15, 135, 30 ), 0x009900 );  
        pEvt.target.loader.unload();  
    }  
}
```

Aussitôt le contenu supprimé, l'événement `Event.UNLOAD` est diffusé.

En cas d'erreur de chargement, nous devons gérer et indiquer à l'utilisateur qu'une erreur s'est produite. Pour cela nous devons obligatoirement écouter l'événement `IOErrorEvent.IO_ERROR`.

Ce dernier est diffusé lorsque le contenu n'a pu être chargé, pour des raisons d'accès ou de compatibilité. Si nous tentons de charger un contenu non géré et que l'événement `IOErrorEvent.IO_ERROR` n'est pas écouté, une erreur à l'exécution est levée.

---

Attention, si le bouton retour du navigateur est cliqué pendant le chargement de contenu externe par le lecteur Flash. L'événement `IOErrorEvent.IO_ERROR` est diffusé, alors que le contenu est pourtant compatible ou disponible.

---

Il convient donc de *toujours* écouter cet événement.

---

Dans le code suivant, un fichier non compatible est chargé, l'événement `IOErrorEvent.IO_ERROR` est diffusé :

```
// création du chargeur
var chargeur:Loader = new Loader();

// référence à l'objet LoaderInfo
var cli:LoaderInfo = chargeur.contentLoaderInfo;

// écoute des événements liés au chargement
cli.addEventListener ( Event.OPEN, debutChargement );
cli.addEventListener ( Event.INIT, initialisation );
cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
cli.addEventListener ( Event.COMPLETE, chargementTermine );
cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );
cli.addEventListener ( HTTPStatusEvent.HTTP_STATUS, echecHTTP );
cli.addEventListener ( Event.UNLOAD, suppressionContenu );

// tentative de chargement d'un document PDF
// entraîne la diffusion de l'événement IOErrorEvent.IO_ERROR
var maRequete:URLRequest = new URLRequest ("document.pdf");

// chargement du contenu
chargeur.load( maRequete );

// ajout à la liste d'affichage
addChild ( chargeur );

function debutChargement ( pEvt:Event ):void
{
    trace( pEvt );
}

function initialisation ( pEvt:Event ):void
{

```

```
        trace( pEvt );
    }
    function chargement ( pEvt:ProgressEvent ):void
    {
        trace( pEvt );
    }
    function chargementTermine ( pEvt:Event ):void
    {
        trace( pEvt );
    }
    function suppressionContenu ( pEvt:Event ):void
    {
        trace( pEvt );
    }
    function echecChargement ( pEvt:Event ):void
    {
        // affiche : [IOErrorEvent type="ioError" bubbles=false cancelable=false
        // eventPhase=2 text="Error #2124: Le type du fichier chargé est inconnu. URL:
        // file:///D:/Work/Bytearray.org/O%27Reilly/Pratique%20d%27ActionScript/Chapitre
        // %2013/flas/document.pdf"]
        trace ( pEvt );
    }
}
```

Lorsqu'une animation est chargée par un objet `Loader`, l'objet `LoaderInfo` active de nouvelles propriétés liées au SWF.

Voici le détail de chacune d'entre elles :

- `LoaderInfo.frameRate` : Indique la cadence du SWF chargé.
- `LoaderInfo.applicationDomain` : retourne une référence à l'objet `ApplicationDomain` contenant les définitions de classe du SWF chargé.
- `LoaderInfo.actionScriptVersion` : indique la version d'ActionScript du SWF chargé.
- `LoaderInfo.parameters` : objet contenant les FlashVars associés au SWF chargé.
- `LoaderInfo.swfVersion` : indique la version du SWF chargé.

Toute tentative d'accès, à l'une de ces propriétés lorsqu'une image est chargée, se traduit par un échec. Dans le code suivant, une image est chargée, nous tentons alors d'accéder à la propriété `frameRate` :

```
function chargementTermine ( pEvt:Event ):void
{
    // tentative d'accès à la cadence de l'élément chargé
    trace( pEvt.target.frameRate );
}
```

L'erreur à l'exécution suivante est levée :

```
Error: Error #2098: L'objet en cours de chargement n'est pas un fichier .swf,
vous ne pouvez donc pas en extraire des propriétés SWF.
```

Afin d'utiliser ces propriétés nous chargeons une animation :

```
// url à atteindre
var maRequete:URLRequest = new URLRequest ("animation.swf");
```

Puis nous accédons au sein de la fonction `chargementTermine` aux différentes informations liées au SWF :

```
function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // affiche : 30
    trace( objetLoaderInfo.frameRate );

    // affiche : [object ApplicationDomain]
    trace( objetLoaderInfo.applicationDomain );

    // affiche : 3
    trace( objetLoaderInfo.actionScriptVersion );

    // affiche : [object Object]
    trace( objetLoaderInfo.parameters );

    // affiche : 9
    trace( objetLoaderInfo.swfVersion );
}
```

Attention, il convient de toujours utiliser les propriétés `width` et `height` de l'objet `LoaderInfo` afin de récupérer la largeur et hauteur du contenu chargé. Dans le cas de chargement d'animations, la propriété `content` référence le scénario principal du SWF.

Ainsi, l'utilisation des propriétés `width` et `height` renvoient la taille du contenu de l'animation, et non les dimensions du SWF :

```
function chargementTermine ( pEvt:Event ):void
```

---

```
{  
  
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );  
  
    var contenu:DisplayObject = objetLoaderInfo.content;  
  
    // accède aux dimensions du contenu de l'animation  
    // affiche : 187.5 187.5  
    trace( contenu.width, contenu.height );  
  
    // accède aux dimensions du SWF  
    // affiche : 550 400  
    trace( objetLoaderInfo.width, objetLoaderInfo.height );  
  
}
```

Nous pouvons tester lors de la fin du chargement le type de contenu chargé à l'aide du mot clé `is` afin de déterminer si le contenu chargé est une animation ou une image

En testant le type de la propriété `content` nous pouvons déterminer le type de contenu chargé :

```
function chargementTermine ( pEvt:Event ):void  
{  
  
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );  
  
    var contenu:DisplayObject = objetLoaderInfo.content;  
  
    if ( contenu is Bitmap )  
    {  
  
        trace("Une image a été chargée !");  
  
    } else  
    {  
  
        trace("Une animation a été chargée !");  
  
    }  
  
}
```

Nous venons de traiter l'utilisation de la classe `LoaderInfo` à travers la propriété `contentLoaderInfo` de l'objet `Loader`. A l'inverse lorsque nous accédons à la propriété `loaderInfo` d'un `DisplayObject` nous accédons aux informations du SWF contenant le `DisplayObject`.

A l'instar des propriétés `stage` et `root`, la propriété `loaderInfo` renvoie `null`, tant que l'objet graphique n'est pas présent au sein de la liste d'affichage :



```
// création d'un objet graphique
var monSprite:Sprite = new Sprite();

// affiche : null
trace( monSprite.loaderInfo );

// ajout à la liste d'affichage
addChild ( monSprite );

// affiche : [object LoaderInfo]
trace( monSprite.loaderInfo );

// affiche : true
trace( loaderInfo == monSprite.loaderInfo );
```

Afin d'accéder aux différentes propriétés de la classe `LoaderInfo`. Nous devons attendre la fin du chargement du SWF en cours. Si nous tentons d'accéder aux propriétés de la classe `LoaderInfo` avant l'événement `Event.INIT` :

```
trace( monSprite.loaderInfo.width );
```

L'erreur suivante est levée à l'exécution :

```
Error: Error #2099: L'objet en cours de chargement n'est pas suffisamment
chargé pour fournir ces informations.
```

A l'aide de l'événement `Event.INIT` ou `Event.COMPLETE` nous pouvons accéder correctement aux informations du SWF en cours :

```
// création d'un objet graphique
var monSprite:Sprite = new Sprite();

// ajout à la liste d'affichage
addChild ( monSprite );

// écoute de l'événement Event.COMPLETE auprès de
// l'objet LoaderInfo du SWF en cours
monSprite.loaderInfo.addEventListener ( Event.COMPLETE, chargementTermine );

function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // affiche : [object LoaderInfo]
    trace( objetLoaderInfo );

    // affiche : 550 400
    trace( objetLoaderInfo.width, objetLoaderInfo.height );

    // affiche : application/x-shockwave-flash
    trace( objetLoaderInfo.contentType );

    // affiche : 12
    trace( objetLoaderInfo.frameRate );

    // affiche : 577 577
    trace ( objetLoaderInfo.bytesLoaded, objetLoaderInfo.bytesTotal );
```

```
| }
```

Nous pouvons aussi référencer l'objet `LoaderInfo` par la propriété `loaderInfo` du scénario principal :

```
// écoute de l'événement Event.COMPLETE auprès de
// l'objet LoaderInfo du SWF en cours
loaderInfo.addEventListener ( Event.COMPLETE, chargementTermine );
```

De cette manière, un SWF peut gérer son préchargement de manière autonome. Ainsi, chaque SWF possède un objet `LoaderInfo` associé regroupant les informations relatives à celui-ci.

Nous retrouvons avec le code précédent, l'équivalent de l'événement `onLoad` existant en ActionScript 1 et 2.

## A retenir

- La classe `LoaderInfo` contient des informations relatives à un élément chargé de type bitmap ou SWF.
- La propriété `contentLoaderInfo` de l'objet `Loader` référence l'objet `LoaderInfo` associé à l'élément chargé.
- Tous les objets de type `DisplayObject` possèdent une propriété `loaderInfo` associé au SWF en cours.
- Lorsqu'une image est chargée, seules certaines propriétés de la classe `LoaderInfo` sont utilisables.
- Lorsqu'un SWF est chargé, la totalité des propriétés de la classe `LoaderInfo` sont utilisables.

## Interagir avec le contenu

Il est généralement courant, de devoir interagir avec le contenu chargé. La classe `Loader` est un `DisplayObjectContainer` et hérite donc de la classe `InteractiveObject` facilitant l'interactivité souris et clavier. Il est cependant impossible d'activer la propriété `buttonMode` auprès de l'objet `Loader`, l'interactivité est donc réduite.

Une première technique consiste à déplacer le contenu de l'objet `Loader` à l'aide de la méthode `addChild`.

Dans le code suivant, nous accédons au contenu chargé puis nous le déplaçons au sein d'un autre conteneur :

```
function chargementTermine ( pEvt:Event ):void
{
    // référence le contenu
    var contenu:DisplayObject = pEvt.target.content;
```

```
// affiche : [object Loader]
trace( contenu.parent );

// réattribution de conteneur, l'image quitte l'objet Loader
addChild ( contenu );

// affiche : [object MainTimeline]
trace( contenu.parent );

}
```

Nous référençons le contenu à l'aide de la propriété `content` de l'objet `LoaderInfo`. Il convient de rester le plus générique possible, nous utilisons donc le type commun `DisplayObject` propre à tout type de contenu chargé, puis nous déplaçons le contenu chargé à l'aide de la méthode `addChild`.

---

Souvenez-vous, lorsqu'un `DisplayObject` déjà présent dans la liste d'affichage est passé à la méthode `addChild`, l'objet sur lequel la méthode est appelée devient le nouveau conteneur.

C'est le concept de réattribution de parent couvert au sein du chapitre 4 intitulé *Liste d'affichage*.

---

Afin de dupliquer l'image chargée, nous n'avons pas à recharger l'image au sein d'un autre objet `Loader`. Nous copions les données bitmap puis les associons à un nouvel objet `Bitmap` :

```
function chargementTermine ( pEvt:Event ):void
{
    // référence le contenu
    var contenu:DisplayObject = pEvt.target.content;

    // si le contenu chargé est une image
    if ( contenu is Bitmap )
    {
        // transtype en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        var donneesBitmap:BitmapData = image.bitmapData;

        // création d'une copie des données bitmaps
        var copieDonneesBitmap:BitmapData = donneesBitmap.clone();

        var copieImage:Bitmap = new Bitmap ( copieDonneesBitmap );

        copieImage.x = image.width + 5;

        addChild ( image );
        addChild ( copieImage );
    }
}
```

```

    }
}

```

La figure 13-3 illustre le résultat :



*Figure 13-3. Copie de l'image chargée.*

Afin de rendre les images cliquables, nous les ajoutons au sein d'objets interactifs tels `Sprite` :

```

function chargementTermine ( pEvt:Event ):void
{
    // référence le contenu
    var contenu:DisplayObject = pEvt.target.content;

    // si le contenu chargé est une image
    if ( contenu is Bitmap )
    {
        // transtypage en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        var premierConteneur:Sprite = new Sprite();
        premierConteneur.buttonMode = true;
        premierConteneur.addChild( image );

        var donneesBitmap:BitmapData = image.bitmapData;
    }
}

```

```
// création d'une copie des données bitmaps
var copieDonneesBitmap:BitmapData = donneesBitmap.clone();

var copieImage:Bitmap = new Bitmap ( copieDonneesBitmap );

var secondConteneur:Sprite = new Sprite();

secondConteneur.buttonMode = true;

secondConteneur.addChild( copieImage );

secondConteneur.x = premierConteneur.width + 5;

addChild ( premierConteneur );
addChild ( secondConteneur );

premierConteneur.addEventListener( MouseEvent.CLICK, clicImage );
secondConteneur.addEventListener( MouseEvent.CLICK, clicImage );

}

}

function clicImage ( pEvt:MouseEvent ):void
{

    // affiche : [MouseEvent type="click" bubbles=true cancelable=false
    eventPhase=2 localX=48 localY=36 stageX=220 stageY=36 relatedObject=null
    ctrlKey=false altKey=false shiftKey=false delta=0]
    trace( pEvt );

}
```

Les deux images bitmaps sont placées dans un conteneur `Sprite` spécifique, facilitant l'interactivité. Afin d'activer le comportement bouton, nous utilisons la propriété `buttonMode`.

Souvenez-vous, afin de rendre plus souple notre code, nous capturons l'événement `MouseEvent.CLICK` auprès du parent des événements réactifs. Nous préférons donc le code suivant :

```
// création d'un conteneur pour les images
var conteneurImages:Sprite = new Sprite();

addChild ( conteneurImages );

// capture de l'événement MouseEvent.MOUSE_CLICK auprès du conteneur
conteneurImages.addEventListener( MouseEvent.CLICK, clicImages, true );

function clicImages ( pEvt:MouseEvent ):void
{

    // [MouseEvent type="click" bubbles=true cancelable=false eventPhase=1
    localX=87 localY=4 stageX=259 stageY=4 relatedObject=null ctrlKey=false
    altKey=false shiftKey=false delta=0]
    trace( pEvt );

}
```

```
function chargementTermine ( pEvt:Event ):void
{
    // référence le contenu
    var contenu:DisplayObject = pEvt.target.content;

    // si le contenu chargé est une image
    if ( contenu is Bitmap )
    {
        // transtype en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        var premierConteneur:Sprite = new Sprite();

        premierConteneur.buttonMode = true;

        premierConteneur.addChild( image );

        var donneesBitmap:BitmapData = image.bitmapData;

        // création d'une copie des données bitmaps
        var copieDonneesBitmap:BitmapData = donneesBitmap.clone();

        var copieImage:Bitmap = new Bitmap ( copieDonneesBitmap );

        var secondConteneur:Sprite = new Sprite();

        secondConteneur.buttonMode = true;

        secondConteneur.addChild( copieImage );

        secondConteneur.x = premierConteneur.width + 5;

        conteneurImages.addChild ( premierConteneur );
        conteneurImages.addChild ( secondConteneur );
    }
}
```

Grâce à la capture de l'événement `MouseEvent.CLICK`, lorsque de nouvelles images seront placées ou supprimées du conteneur `conteneurImages`, aucune nouvelle souscription ou désinscription d'écouteurs ne sera nécessaire.

A chaque clic sur les images nous la supprimons de la liste d'affichage, pour cela nous modifions la fonction `clicImages` :

```
function clicImages ( pEvt:MouseEvent ):void
{
    pEvt.currentTarget.removeChild ( DisplayObject ( pEvt.target ) );
}
```

Lorsque l'image est cliquée, celle-ci est supprimée de la liste d'affichage. Une référence vers l'image d'origine demeure cependant au sein de l'objet `Loader`, il nous est donc impossible de libérer correctement les ressources.

De plus, en déplaçant le contenu de l'objet `Loader` nous modifions son fonctionnement interne. Ainsi, lors du prochain appel de la méthode `load`, celui-ci tentera de supprimer son contenu afin d'accueillir le nouveau. Celui-ci ayant été déplacé, l'objet `Loader` tente alors de supprimer un objet qui n'est plus son enfant, ce qui provoque alors une erreur à l'exécution.

---

Ce comportement est un bug du lecteur Flash 9.0.115  
qui sera corrigé au sein du prochain lecteur Flash 10.

---

Il est donc fortement déconseillé de déplacer le contenu de l'objet `Loader` sous peine de perturber le mécanisme interne de celui-ci. De manière générale nous préférons créer un objet `Loader` pour chaque contenu chargé.

Nous pouvons donc établir le bilan suivant :

```
// bonne pratique
addChild ( chargeur );

// mauvaise pratique
addChild ( chargeur.content );
```

Dans le code suivant, nous chargeons plusieurs images au sein de différents objets `Loader` :

```
var conteneurImages:Sprite = new Sprite();

addChild ( conteneurImages );

// capture de l'événement MouseEvent.CLICK auprès du conteneur
conteneurImages.addEventListener( MouseEvent.CLICK, clicImages, true );

var chargeurImage:Loader;

var requete:URLRequest = new URLRequest();

var tableauImages:Array = new Array ( "photo1.jpg", "photo2.jpg",
"photo3.jpg", "photo4.jpg", "photo5.jpg", "photo3.jpg" );

var lng:int = tableauImages.length;

var colonnes:int = 4;

for ( var i:int = 0; i< lng; i++ )
{

    // création du chargeur
    chargeurImage = new Loader();
```

```
    requete.url = tableauImages[i];

    chargeurImage.load ( requete );

    chargeurImage.x = Math.round (i % colonnes) * 120;
    chargeurImage.y = Math.floor (i / colonnes) * 80

    conteneurImages.addChild ( chargeurImage );
}

function clicImages ( pEvt:MouseEvent ):void
{
    // [MouseEvent type="click" bubbles=true cancelable=false eventPhase=1
    // localX=10 localY=49 stageX=250 stageY=49 relatedObject=null ctrlKey=false
    // altKey=false shiftKey=false delta=0]
    trace( pEvt );
}
```

Afin de rendre les images cliquables, nous les imbriquons dans des conteneurs de type **Sprite** :

```
var conteneurImages:Sprite = new Sprite();

addChild ( conteneurImages );

// capture de l'événement MouseEvent.MOUSE_CLICK auprès du conteneur
conteneurImages.addEventListener( MouseEvent.CLICK, clicImages, true );

var chargeurImage:Loader;

var conteneurLoader:Sprite;

var requete:URLRequest = new URLRequest();

var tableauImages:Array = new Array ("photo1.jpg", "photo2.jpg",
"photo3.jpg", "photo4.jpg", "photo5.jpg", "photo3.jpg");

var lng:int = tableauImages.length;

var colonnes:int = 4;

for (var i:int = 0; i< lng; i++ )
{
    // création du chargeur
    chargeurImage = new Loader();

    conteneurLoader = new Sprite();

    conteneurLoader.addChild( chargeurImage );

    requete.url = tableauImages[i];

    chargeurImage.load ( requete );

    conteneurLoader.x = Math.round (i % colonnes) * 120;
```



```

    conteneurLoader.y = Math.floor (i / colonnes) * 80

    conteneurImages.addChild ( conteneurLoader );

}

function clicImages ( pEvt:MouseEvent ):void
{
    // [MouseEvent type="click" bubbles=true cancelable=false eventPhase=1
    localX=10 localY=49 stageX=250 stageY=49 relatedObject=null ctrlKey=false
    altKey=false shiftKey=false delta=0]
    trace( pEvt );
}

```

L'activation du mode bouton est rendue possible grâce à l'activation de la propriété `buttonMode` sur les instances de `Sprite` contenant les objets `Loader` :

```

    conteneurLoader.buttonMode = true;

```

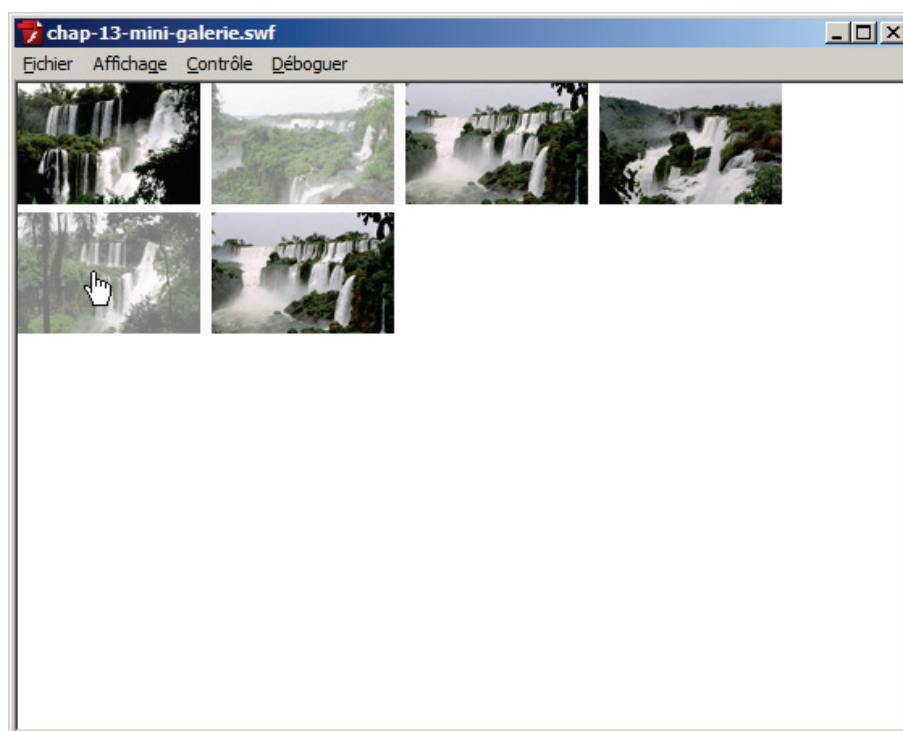
Puis nous modifions la fonction `clicImages` :

```

function clicImages ( pEvt:MouseEvent ):void
{
    DisplayObject( pEvt.target ).alpha = .5;
}

```

La figure 13-4 illustre le résultat :



*Figure 13-4. Mini galerie.*

A chaque image cliquée, celle-ci passe en opacité réduite afin de mémoriser visuellement l'action utilisateur.

## A retenir

- Le contenu chargé est accessible par la propriété `content` de l'objet `LoaderInfo`.
- Bien qu'étant une sous-classe de `DisplayObjectContainer`, la classe `Loader` ne dispose pas de toutes les capacités propres à ce type.
- Il est techniquement possible de déplacer le contenu chargé et de réattribuer son parent, mais il est fortement recommandé de conserver le contenu au sein de l'objet `Loader`.
- Afin d'afficher le curseur main au survol d'un objet `Loader`, nous devons au préalable l'imbriquer au sein d'un conteneur de type `Sprite`.

## Créer une galerie

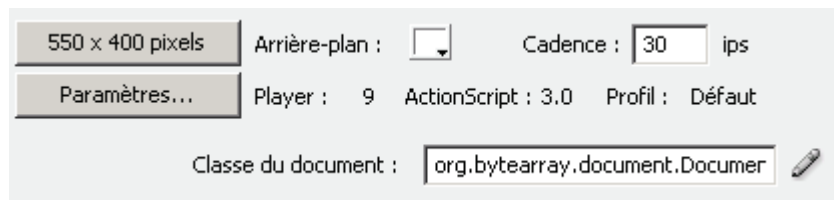
Nous allons mettre en pratique un ensemble de notions abordées depuis le début de l'ouvrage ainsi que celles abordées en début de ce chapitre. Nous allons créer une galerie photo et ainsi voir comment la concevoir de manière optimisée.

Ouvrez un nouveau document Flash CS3, puis définissez une classe de document au sein d'un paquetage `org.bytearray.document` :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    public class Document extends ApplicationDefault
    {
        public function Document ()
        {
            trace( this );
        }
    }
}
```

Cette classe étend la classe `ApplicationDefault` créée au cours du chapitre 11 intitulé *Classe du document*, nous récupérerons ainsi toutes les fonctionnalités définies par celle-ci.

Une fois créée, nous définissons celle-ci comme classe du document grâce au panneau approprié de l'inspecteur de propriétés :



*Figure 13-5. Classe du document.*

Nous définissons la classe `ChargeurMedia` au sein du paquetage `org.bytearray.chargement`. Cette classe va nous permettre de gérer facilement le chargement d'images. Nous pourrions utiliser simplement la classe `Loader` mais nous avons besoin d'augmenter les ses fonctionnalités.

La classe `ChargeurMedia` se chargera de toute la logique de chargement et de redimensionnement des médias chargés. Ainsi pour chaque projet nécessitant un chargement de contenu notre classe `ChargeurMedia` pourra être réutilisée.

Voici le code de base de la classe `ChargeurMedia` :

```
package org.bytearray.chargement

{

    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.display.LoaderInfo;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;

    public class ChargeurMedia extends Sprite
    {

        private var chargeur:Loader;
        private var cli:LoaderInfo;

        public function ChargeurMedia ()
        {

            chargeur = new Loader();

            addChild ( chargeur );
        }
    }
}
```

```
        cli = chargeur.contentLoaderInfo;

        cli.addEventListener ( Event.INIT, initChargement );
        cli.addEventListener ( Event.OPEN, démarreChargement );
        cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
        cli.addEventListener ( Event.COMPLETE, chargementTermine );
        cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );

    }

    private function démarreChargement ( pEvt:Event ):void
    {

        trace ("démarre chargement");

    }

    private function initChargement ( pEvt:Event ):void
    {

        trace ("initialisation du chargement");

    }

    private function chargement ( pEvt:Event ):void
    {

        trace ("chargement en cours");

    }

    private function chargementTermine ( pEvt:Event ):void
    {

        trace ("chargement terminé");

    }

    private function echecChargement ( pEvt:Event ):void
    {

        trace ("erreur de chargement");

    }

}

}
```

Une fois la classe **ChargeurMedia** définie, nous pouvons l’instancier au sein de la classe **Document** :

```
package org.bytearray.document
{

    import org.bytearray.abstrait.ApplicationDefaut;
```

```
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefault
{
    // visionneuse
    private var visionneuse:ChargeurMedia;

    public function Document ()
    {
        // création de l'objet Loader
        visionneuse = new ChargeurMedia ();

        addChild ( visionneuse );
    }
}
```

Certains d'entre vous seront peut-être étonnés de voir que nous n'héritons pas de la classe `Loader`. Pourtant, toutes les raisons paraissent ici réunies pour utiliser l'héritage. Notre classe `ChargeurMedia` doit augmenter les capacités de la classe `Loader`, l'héritage semble donc être la meilleure solution.

Nous allons préférer ici une deuxième approche déjà utilisée au cours des chapitres précédents. La classe `ChargeurMedia` ne *sera* pas un objet `Loader` mais *possèdera* un objet `Loader`.

Vous souvenez-vous de cette opposition ?

## La composition

Nous allons préférer ici l'utilisation de la *composition* à *l'héritage*. Nous avons vu au cours du chapitre 8 intitulé *Programmation orientée objet* les avantages qu'offre la composition en matière d'évolutivité et de modification du code.

Lorsque l'application est exécutée, le constructeur de la classe `Document` instancie un objet `ChargeurMedia`. Très généralement, les données à charger proviennent d'une base de données ou d'un fichier XML. Nous allons réaliser une première version à l'aide d'un tableau contenant les URL des images à charger.

Nous créons ensuite un tableau contenant l'adresse de chaque image :

```
package org.bytearray.document
{
```

```
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefaut
{
    // visionneuse
    private var visionneuse:ChargeurMedia;

    // données
    private var tableauImages:Array;

    public function Document ()
    {
        // tableau contenant les url des images
        tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
        "imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

        // création de l'objet Loader
        visionneuse = new ChargeurMedia ();

        addChild ( visionneuse );
    }
}
```

Un répertoire `imgs` est créé contenant différentes images portant le nom de celles définies dans le tableau `tableauImages`. Pour le moment, la classe `ChargeurMedia` n'est pas utilisable, nous devons ajouter une méthode permettant de charger le media spécifié.

Nous importons les classes `URLRequest` et `LoaderContext` dans la définition de la classe :

```
import flash.display.Loader;
import flash.display.Sprite;
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.events.IOErrorEvent;
import flash.net.URLRequest;
import flash.system.LoaderContext;
```

Puis, nous ajoutons une méthode `load`. Celle-ci appelle en interne la méthode `load` de l'objet `Loader` :

```
public function load ( pURL:URLRequest, pContext:LoaderContext=null ):void
{
    chargeur.load ( pURL, pContext );
}
```

En déléguant les fonctionnalités à une classe interne. La classe `ChargeurMedia` procède à une *délégation*.

---

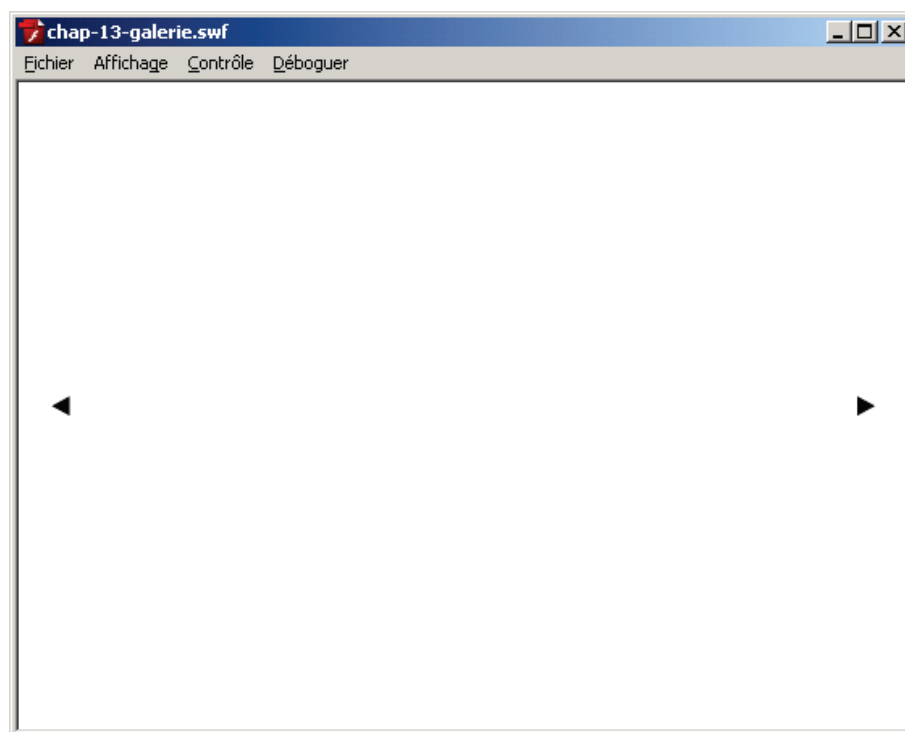
Ce mécanisme est l'un des plus puissants de la composition. Grâce à cela, la classe `ChargeurMedia` peut à l'exécution décider d'utiliser un autre type d'objet dans lequel charger les données.

Chose impossible si la classe `ChargeurMedia` avait héritée de la classe `Loader`. C'est pour cette raison que la composition est considérée comme une approche dynamique, s'opposant au caractère statique de l'héritage lié à la compilation.

---

Sur la scène nous posons deux boutons `boutonSuivant` et `boutonPrecedent`, auxquels nous donnons deux noms d'occurrences respectifs.

La figure 13-6 illustre l'interface :



*Figure 13-6. Boutons de navigation.*

Nous définissons chaque bouton au sein de la classe `Document` :

```
package org.bytearray.document
{
```

```
import flash.display.SimpleButton;
import flash.events.MouseEvent;
import org.bytearray.abstrait.ApplicationDefault;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefault
{
    // interface
    public var boutonSuivant:SimpLeButton;
    public var boutonPrecedent:SimpLeButton;

    // visionneuse
    private var visionneuse:ChargeurMedia;

    // données
    private var tableauImages:Array;

    public function Document ()
    {
        // tableau contenant les url des images
        tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

        // création de l'objet Loader
        visionneuse = new ChargeurMedia ();

        addChild ( visionneuse );

        boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
        boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );
    }

    private function clicPrecedent ( pEvt:MouseEvent=null ):void
    {
        trace("Image précédente");
    }

    private function clicSuivant ( pEvt:MouseEvent=null ):void
    {
        trace("Image suivante");
    }
}
}
```

Puis nous associons la logique spécifique à chaque bouton :

```
package org.bytearray.document
```

---



```
{

import flash.display.SimpleButton;
import flash.events.MouseEvent;
import flash.net.URLRequest;
import org.bytearray.abstrait.ApplicationDefault;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefault

{

    // interface
    public var boutonSuivant:SimpLeButton;
    public var boutonPrecedent:SimpLeButton;

    // visionneuse
    private var visionneuse:ChargeurMedia;

    // données
    private var tableauImages:Array;
    private var position:int;
    private var requete:URLRequest;

    public function Document ()

    {

        position = -1;

        requete = new URLRequest();

        // tableau contenant les url des images
        tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

        // création de l'objet Loader
        visionneuse = new ChargeurMedia ();

        addChild ( visionneuse );

        boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
        boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

    }

    private function clicPrecedent ( pEvt:MouseEvent=null ):void

    {

        position = Math.max ( 0, --position );

        requete.url = tableauImages [ position ];

        visionneuse.load ( requete );

    }

    private function clicSuivant ( pEvt:MouseEvent=null ):void

    {
```

```
        position = ++position % tableauImages.length;

        requete.url = tableauImages [ position ];

        visionneuse.load ( requete );

    }

}
```

A chaque clic, nous déclenchons la fonction `clicPrecedent` ou `clicSuivant`. Nous remarquons que lorsque nous cliquons sur le bouton `boutonSuivant` nous tournons en boucle, à l'inverse le `boutonPrecedent` bute à l'index 0.

En créant une propriété `position`, nous incrémentons ou décrémentons celle-ci, afin de naviguer au sein du tableau `tableauImages`.

Ne vous est-il jamais arrivé de charger des images de différentes dimensions, et de souhaiter que quelles que soient leurs tailles, celles-ci s'affiche correctement dans un espace donné ?

## Redimensionnement automatique

L'une des premières fonctionnalités que nous allons ajouter à la classe `Loader` concerne l'ajustement automatique de l'image selon une dimension spécifiée.

Les images chargées au sein de la galerie peuvent parfois déborder ou ne pas être ajustées automatiquement. Cela est généralement prévu par la partie serveur mais afin de sécuriser notre application, nous allons intégrer ce mécanisme côté client. Cela nous évitera de mauvaises surprises au cas où l'image n'aurait pas été correctement redimensionnée au préalable.

Lorsque l'événement `Event.COMPLETE` est diffusé, nous ajustons automatiquement la taille de l'objet `ChargeurMedia` aux dimensions voulues tout en conservant les proportions afin de ne pas déformer le contenu chargé :

```
package org.bytearray.chargement

{

    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.display.LoaderInfo;
    import flash.events.Event;
    import flash.events.ProgressEvent;
```

```
import flash.events.IOErrorEvent;
import flash.net.URLRequest;
import flash.system.LoaderContext;
import flash.display.DisplayObject;

public class ChargeurMedia extends Sprite

{

    private var chargeur:Loader;
    private var cli:LoaderInfo;
    private var largeur:Number;
    private var hauteur:Number;

    public function ChargeurMedia ( pLargeur:Number, pHauteur:Number )

    {

        largeur = pLargeur;

        hauteur = pHauteur;

        chargeur = new Loader();

        addChild ( chargeur );

        cli = chargeur.contentLoaderInfo;

        cli.addEventListener ( Event.INIT, initChargement );
        cli.addEventListener ( Event.OPEN, demarreChargement );
        cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
        cli.addEventListener ( Event.COMPLETE, chargementTermine );
        cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );

    }

    public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void

    {

        chargeur.load ( pURL, pContext );

    }

    private function demarreChargement ( pEvt:Event ):void

    {

        trace ("d  marre chargement");

    }

    private function initChargement ( pEvt:Event ):void

    {

        trace ("initialisation chargement");

    }

    private function chargement ( pEvt:Event ):void
```

```
        {  
            trace ("chargement en cours");  
        }  
  
        private function chargementTermine ( pEvt:Event ):void  
        {  
            var contenuCharge:DisplayObject = pEvt.target.content;  
  
            var ratio:Number = Math.min ( largeur / contenuCharge.width,  
hauteur / contenuCharge.height );  
  
            scaleX = scaleY = 1;  
  
            if ( ratio < 1 ) scaleX = scaleY = ratio;  
        }  
  
        private function echecChargement ( pEvt:Event ):void  
        {  
            trace ("erreur de chargement");  
        }  
    }  
}
```

Puis nousinstancions l'objet `ChargeurMedia` en spécifiant les dimensions à respecter :

```
// création de la visionneuse  
visionneuse = new ChargeurMedia ( 350, 450 );
```

Si nous testons la galerie nous remarquons que les images sont bien affichées et correctement redimensionnées si cela est nécessaire.

Pour le moment, elles ne sont pas centrées :



*Figure 13-7. Galerie.*

Nous devons placer le code correspondant au centrage de l'image en dehors de la classe `ChargeurMedia`. Intégrer une telle logique à notre classe `ChargeurMedia` la rendrait rigide.

N'oubliez pas un point essentiel de la programmation orientée objet ! Nous devons isoler ce qui risque d'être modifié d'un projet à un autre, nous plaçons donc le positionnement des images en dehors de la classe `ChargeurMedia`.

Afin de pouvoir utiliser la classe `ChargeurMedia` comme la classe `Loader`, nous devons à présent diffuser tous les événements nécessaires à son bon fonctionnement. Afin de permettre cette diffusion nous devons d'abord les écouter en interne, puis les rediriger par la suite.

Nous modifions pour cela chaque fonction écouteur, puis nous procédons à la redirection de chaque événement :

```
package org.bytearray.chargement
{
    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.display.DisplayObject;
    import flash.display.LoaderInfo;
    import flash.events.Event;
```

```
import flash.events.ProgressEvent;
import flash.events.IOErrorEvent;
import flash.net.URLRequest;
import flash.system.LoaderContext;

public class ChargeurMedia extends Sprite
{
    private var chargeur:Loader;
    private var cli:LoaderInfo;
    private var largeur:Number;
    private var hauteur:Number;

    public function ChargeurMedia ( pLargeur:Number, pHauteur:Number )
    {
        largeur = pLargeur;
        hauteur = pHauteur;
        chargeur = new Loader();
        addChild ( chargeur );
        cli = chargeur.contentLoaderInfo;

        cli.addEventListener ( Event.INIT, initChargement );
        cli.addEventListener ( Event.OPEN, demarreChargement );
        cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
        cli.addEventListener ( Event.COMPLETE, chargementTermine );
        cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );
    }

    public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void
    {
        chargeur.load ( pURL, pContext );
    }

    private function demarreChargement ( pEvt:Event ):void
    {
        dispatchEvent ( pEvt );
    }

    private function initChargement ( pEvt:Event ):void
    {
        dispatchEvent ( pEvt );
    }

    private function chargement ( pEvt:Event ):void
```

```
{  
    dispatchEvent ( pEvt );  
}  
  
private function chargementTermine ( pEvt:Event ):void  
{  
    var contenuCharge:DisplayObject = pEvt.target.content;  
    var ratio:Number = Math.min ( largeur / contenuCharge.width,  
    hauteur / contenuCharge.height );  
    scaleX = scaleY = 1;  
    if ( ratio < 1 ) scaleX = scaleY = ratio;  
    dispatchEvent ( pEvt );  
}  
  
private function echecChargement ( pEvt:Event ):void  
{  
    dispatchEvent ( pEvt );  
}  
}  
}
```

De cette manière, la classe `ChargeurMedia`, diffuse les mêmes événements que la classe `Loader`. Cela reste totalement transparent pour l'utilisateur de la classe `ChargeurMedia` et s'inscrit comme suite logique de la *composition* et de la *délégation*.

---

Lorsqu'ils sont redirigés, les objets événementiels voient leur propriété `target` référencer l'objet diffuseur ayant relayé l'événement. La propriété `target` fait donc désormais référence à l'objet `ChargeurMedia` et non plus à l'objet `LoaderInfo`.

---

En écoutant l'événement `Event.COMPLETE` auprès de la classe `ChargeurMedia`, nous sommes avertis de la fin du chargement du contenu :

```
package org.bytearray.document  
{  
    import flash.display.SimpleButton;  
    import flash.events.Event;  
    import flash.events.MouseEvent;
```

```
import org.bytearray.abstrait.ApplicationDefault;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefault
{
    // interface
    public var boutonSuivant:SimpleButton;
    public var boutonPrecedent:SimpleButton;

    // visionneuse
    private var visionneuse:ChargeurMedia;

    public function Document ()
    {
        // tableau contenant les url des images
        var tableauImages:Array = new Array ( "imgs/photo1.jpg",
"imgs/anim1.swf", "imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

        // création de l'objet Loader
        visionneuse = new ChargeurMedia ( 350, 450 );

        addChild ( visionneuse );

        // écoute de l'événement Event.COMPLETE auprès de la visionneuse
        visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
);

        boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
        boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );
    }

    private function chargementTermine ( pEvt:Event ):void
    {
        // centre la visionneuse
        visionneuse.x = ( stage.stageWidth - visionneuse.width ) / 2;
        visionneuse.y = ( stage.stageHeight - visionneuse.height ) / 2;
    }

    private function clicPrecedent ( pEvt:MouseEvent=null ):void
    {
        position = Math.max ( 0, --position );

        requete.url = tableauImages [ position ];

        visionneuse.load ( requete );
    }

    private function clicSuivant ( pEvt:MouseEvent=null ):void
    {

```



```

        position = ++position % tableauImages.length;

        requete.url = tableauImages [ position ];

        visionneuse.load ( requete );

    }

}

```

Afin de garantir un affichage aligné sur les pixels, il est recommandé de toujours positionner une image sur des coordonnées non flottantes.

Pour cela, nous modifions la fonction écouteur `chargementTermine` en arrondissant les coordonnées de l'image centrée :

```

private function chargementTermine ( pEvt:Event ):void
{
    // centre la visionneuse
    visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) / 2 );
    visionneuse.y = int ( ( stage.stageHeight - visionneuse.height ) / 2 );
}

```

Si nous testons la galerie, les images sont correctement chargées et centrées, comme l'illustre la figure 13-8 :



*Figure 13-8. Galerie centrée.*

Afin de charger la première image, nous pouvons déclencher manuellement la fonction écouteur `clicSuivant` :

```
public function Document ()
{
    // tableau contenant les url des images
    var tableauImages:Array = new Array ( "imgs/photo1.jpg",
    "imgs/photo2.jpg", "imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

    // création de l'objet Loader
    visionneuse = new ChargeurMedia ( tableauImages, 350, 450 );

    addChild ( visionneuse );

    // écoute de l'événement Event.COMPLETE auprès de la visionneuse
    visionneuse.addEventListener ( Event.COMPLETE, chargementTermine );

    boutonPrecedent.addEventListener ( MouseEvent.CLICK, clicPrecedent );
    boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

    clicSuivant();
}
```

Lorsque l'application est lancée, la première image est chargée automatiquement. Nous venons de déclencher ici, manuellement une fonction écouteur liée à l'événement `MouseEvent.CLICK`.

Le code de la classe `ChargeurMedia` peut être optimisé en utilisant une seule fonction écouteur pour rediriger différents types d'événements.

Dans le code suivant la méthode `redirigeEvenement` redirige plusieurs événements :

```
package org.bytearray.chargement
{
    import flash.display.Bitmap;
    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.display.DisplayObject;
    import flash.display.LoaderInfo;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;
    import flash.net.URLRequest;
    import flash.system.LoaderContext;

    public class ChargeurMedia extends Sprite
    {
        private var chargeur:Loader;
```

```
private var cli:LoaderInfo;
private var largeur:Number;
private var hauteur:Number;

public function ChargeurMedia ( pLargeur:Number, pHauteur:Number )
{
    largeur = pLargeur;
    hauteur = pHauteur;
    chargeur = new Loader();
    addChild ( chargeur ) ;

    cli = chargeur.contentLoaderInfo;

    cli.addEventListener ( Event.INIT, redirigeEvenement );
    cli.addEventListener ( Event.OPEN, redirigeEvenement );
    cli.addEventListener ( ProgressEvent.PROGRESS, redirigeEvenement
);
    cli.addEventListener ( Event.COMPLETE, chargementTermine );
    cli.addEventListener ( IOErrorEvent.IO_ERROR, redirigeEvenement
);
}

public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void
{
    chargeur.load ( pURL, pContext );
}

private function redirigeEvenement ( pEvt:Event ):void
{
    dispatchEvent ( pEvt );
}

private function chargementTermine ( pEvt:Event ):void
{
    var contenuCharge:DisplayObject = pEvt.target.content;

    var ratio:Number = Math.min ( largeur / contenuCharge.width,
hauteur / contenuCharge.height );

    scaleX = scaleY = 1;

    if ( ratio < 1 ) scaleX = scaleY = ratio;

    dispatchEvent ( pEvt );
}
```

```
    }  
}
```

Bien entendu, cette technique n'est viable que dans le cas où aucune logique ne doit être ajoutée en interne pour chaque événement diffusé. C'est pour cette raison que l'événement `Event.COMPLETE` est toujours écouté en interne par la méthode écouteur `chargementTermine`, car nous devons ajouter une logique de redimensionnement spécifique.

## Gestion du lissage

Nous allons ajouter une seconde fonctionnalité permettant d'afficher les images chargées de manière lissée. Ainsi, lorsque l'image chargée est redimensionnée, grâce au lissage celle-ci n'est pas crénelée.

Pour cela, nous activons la propriété `smoothing` si le contenu chargé est de type `Bitmap` :

```
package org.bytearray.chargement  
{  
    import flash.display.Bitmap;  
    import flash.display.Loader;  
    import flash.display.Sprite;  
    import flash.display.DisplayObject;  
    import flash.display.LoaderInfo;  
    import flash.events.Event;  
    import flash.events.ProgressEvent;  
    import flash.events.IOErrorEvent;  
    import flash.net.URLRequest;  
    import flash.system.LoaderContext;  
  
    public class ChargeurMedia extends Sprite  
    {  
        private var chargeur:Loader;  
        private var cli:LoaderInfo;  
        private var largeur:Number;  
        private var hauteur:Number;  
        private var lissage:Boolean;  
  
        public function ChargeurMedia ( pLargeur:Number, pHauteur:Number,  
pLissage:Boolean=true )  
        {  
            largeur = pLargeur;  
            hauteur = pHauteur;  
            lissage = pLissage;  
            chargeur = new Loader();  
        }  
    }  
}
```

```

        addChild ( chargeur ) ;

        cli = chargeur.contentLoaderInfo;

        cli.addEventListener ( Event.INIT, redirigeEvenement );
        cli.addEventListener ( Event.OPEN, redirigeEvenement );
        cli.addEventListener ( ProgressEvent.PROGRESS, redirigeEvenement
    );
        cli.addEventListener ( Event.COMPLETE, chargementTermine );
        cli.addEventListener ( IOErrorEvent.IO_ERROR, redirigeEvenement
    );

    }

    public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void
    {
        chargeur.load ( pURL, pContext );
    }

    private function redirigeEvenement ( pEvt:Event ):void
    {
        dispatchEvent ( pEvt );
    }

    private function chargementTermine ( pEvt:Event ):void
    {
        var contenuCharge:DisplayObject = pEvt.target.content;

        if ( contenuCharge is Bitmap ) Bitmap ( contenuCharge ).smoothing
= lissage;

        var ratio:Number = Math.min ( largeur / contenuCharge.width,
hauteur / contenuCharge.height );

        scaleX = scaleY = 1;

        if ( ratio < 1 ) scaleX = scaleY = ratio;

        dispatchEvent ( pEvt );
    }
}

```

Puis nous modifions l’instanciation de l’objet **ChargeurMedia** :

```

// création de l'objet Loader
visionneuse = new ChargeurMedia ( tableauImages, 350, 450, true );

```

Les images sont désormais lissées automatiquement. Si nous souhaitons désactiver le lissage, nous le précisons lors de l’instanciation de l’objet `ChargeurMedia` :

```
// création de l'objet Loader
visionneuse = new ChargeurMedia ( tableauImages, 350, 450, false );
```

Souvenez-vous, nous devons toujours intégrer un mécanisme de **désactivation** des objets. Ainsi la classe `ChargeurMedia` ne déroge pas à la règle et intègre un mécanisme de désactivation. Aussitôt supprimée de la liste d’affichage, l’instance de `ChargeurMedia` est désactivée grâce à l’événement `Event.REMOVED_FROM_STAGE`.

Nous utilisons l’événement `Event.ADDED_TO_STAGE` afin d’initialiser l’objet `ChargeurMedia` lorsque celui-ci est affiché :

```
package org.bytearray.chargement

{

    import flash.display.Bitmap;
    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.display.DisplayObject;
    import flash.display.LoaderInfo;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;
    import flash.net.URLRequest;
    import flash.system.LoaderContext;

    public class ChargeurMedia extends Sprite

    {

        private var chargeur:Loader;
        private var cli:LoaderInfo;
        private var largeur:Number;
        private var hauteur:Number;
        private var lissage:Boolean;

        public function ChargeurMedia ( pLargeur:Number, pHauteur:Number,
pLissage:Boolean=true )

        {

            largeur = pLargeur;

            hauteur = pHauteur;

            lissage = pLissage;

            chargeur = new Loader();

            addChild ( chargeur );

            cli = chargeur.contentLoaderInfo;
```

```
        addEventListener ( Event.ADDED_TO_STAGE, activation );
        addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

    }

    private function activation ( pEvt:Event ):void
    {

        cli.addEventListener ( Event.INIT, redirigeEvenement );
        cli.addEventListener ( Event.OPEN, redirigeEvenement );
        cli.addEventListener ( ProgressEvent.PROGRESS, redirigeEvenement
    );

        cli.addEventListener ( Event.COMPLETE, chargementTermine );
        cli.addEventListener ( IOErrorEvent.IO_ERROR, redirigeEvenement
    );

    }

    private function desactivation ( pEvt:Event ):void
    {

        cli.removeEventListener ( Event.INIT, redirigeEvenement );
        cli.removeEventListener ( Event.OPEN, redirigeEvenement );
        cli.removeEventListener ( ProgressEvent.PROGRESS,
redirigeEvenement );
        cli.removeEventListener ( Event.COMPLETE, chargementTermine );
        cli.removeEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

    }

    public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void
    {

        chargeur.load ( pURL, pContext );

    }

    private function redirigeEvenement ( pEvt:Event ):void
    {

        dispatchEvent ( pEvt );

    }

    private function chargementTermine ( pEvt:Event ):void
    {

        var contenuCharge:DisplayObject = pEvt.target.content;

        if ( contenuCharge is Bitmap ) Bitmap ( contenuCharge ).smoothing
= lissage;

        var ratio:Number = Math.min ( largeur / contenuCharge.width,
hauteur / contenuCharge.height );
```

```
        scaleX = scaleY = 1;

        if ( ratio < 1 ) scaleX = scaleY = ratio;

        dispatchEvent ( pEvt );

    }

}
```

La classe `ChargeurMedia` fonctionne sans problème. Nous verrons au cours du chapitre 14 intitulé *Chargement de données* comment remplacer les données par un flux XML chargé dynamiquement.

Nous verrons comment réutiliser la classe `ChargeurMedia` tout en la faisant communiquer avec une classe permettant le chargement de données XML externe.

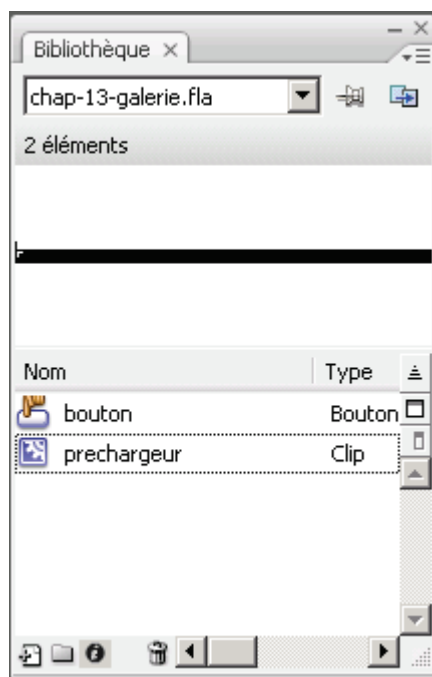
## Précharger le contenu

Il est impératif de précharger tout contenu dynamique. Un utilisateur interrompt très rapidement sa navigation si aucun élément n'indique visuellement l'état du chargement des données.

Dans le cas de notre galerie photo, nous devons précharger chaque image afin d'informer l'utilisateur de l'état du chargement. Nous allons utiliser une barre de préchargement classique puis utiliser chacun des événements afin de synchroniser son affichage.

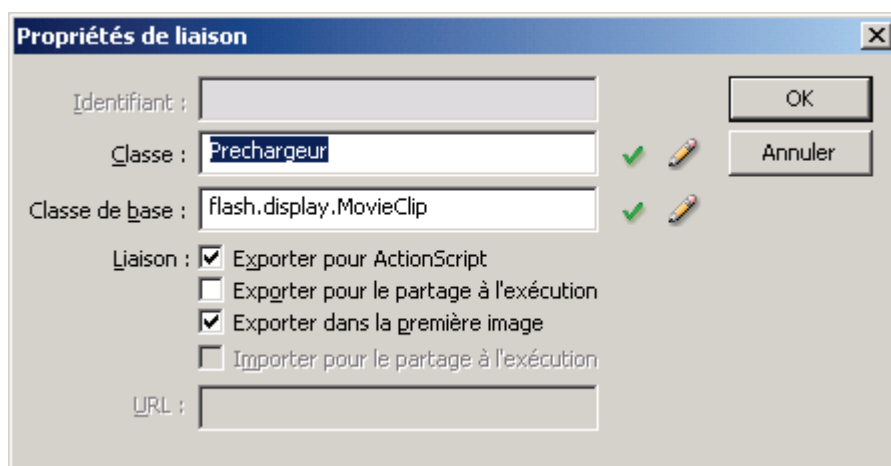
Nous allons tout d'abord créer un symbole contenant la barre de préchargement. La figure 13-9 illustre le symbole en bibliothèque :





*Figure 13-9. Symbole prechargeur.*

Puis nous définissons une classe `Prechargeur` auquel le symbole est lié :



*Figure 13-10. Classe `Prechargeur` associée.*

Une fois le symbole associé, nous l’instancions dès l’initialisation de l’application. Vous remarquerez que celui-ci n’est pas affiché pour l’instant :

```
package org.bytearray.document
{
    import flash.display.SimpleButton;
    import flash.display.Sprite;
```

```
import flash.events.MouseEvent;
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.net.URLRequest;
import org.bytearray.abstrait.ApplicationDefault;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefault
{
    // interface
    public var boutonSuivant:SimpleButton;
    public var boutonPrecedent:SimpleButton;
    public var prechargeur:Prechargeur;

    // visionneuse
    private var visionneuse:ChargeurMedia;

    // données
    private var tableauImages:Array;
    private var position:int;
    private var requete:URLRequest;

    public function Document ()
    {
        position = -1;

        requete = new URLRequest();

        // tableau contenant les url des images
        tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

        // création de l'objet Loader
        visionneuse = new ChargeurMedia ( 445, 335 );

        addChild ( visionneuse );

        prechargeur = new Prechargeur();

        prechargeur.x = Math.round ( (stage.stageWidth -
prechargeur.width) / 2);
        prechargeur.y = Math.round ( (stage.stageHeight -
prechargeur.height) / 2);

        visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
    );

        boutonPrecedent.addEventListener ( MouseEvent.CLICK,
    clicPrecedent );
        boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

        clicSuivant();
    }

    private function chargementTermine ( pEvt:Event ):void
    {

```

```
        // centre la visionneuse
        visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) /
2);
        visionneuse.y = int ( ( stage.stageHeight - visionneuse.height )
/ 2);
    }

    private function clicPrecedent ( pEvt:MouseEvent=null ):void
    {
        position = Math.max ( 0, --position );
        requete.url = tableauImages [ position ];
        visionneuse.load ( requete );
    }

    private function clicSuivant ( pEvt:MouseEvent=null ):void
    {
        position = ++position % tableauImages.length;
        requete.url = tableauImages [ position ];
        visionneuse.load ( requete );
    }
}
}
```

Puis nous écoutons chacun des événements nécessaires au préchargement :

```
package org.bytearray.document
{
    import flash.display.SimpleButton;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.net.URLRequest;
    import org.bytearray.abstrait.ApplicationDefault;
    import org.bytearray.chargement.ChargeurMedia;

    public class Document extends ApplicationDefault
    {
        // interface
        public var boutonSuivant:S.SimpleButton;
        public var boutonPrecedent:S.SimpleButton;
        public var prechargeur:Prechargeur;
```

```
// visionneuse
private var visionneuse:ChargeurMedia;

// données
private var tableauImages:Array;
private var position:int;
private var requete:URLRequest;

public function Document ()

{

    position = -1;

    requete = new URLRequest();

    // tableau contenant les url des images
    tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

    // création de l'objet Loader
    visionneuse = new ChargeurMedia ( 445, 335 );

    addChild ( visionneuse );

    prechargeur = new Prechargeur();

    prechargeur.x = Math.round ( (stage.stageWidth -
prechargeur.width) / 2);
    prechargeur.y = Math.round ( (stage.stageHeight -
prechargeur.height) / 2);

    visionneuse.addEventListener ( Event.OPEN, chargementDemarre );
    visionneuse.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
    visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
);

    boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
    boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

    clicSuivant ( new MouseEvent ( MouseEvent.CLICK ) );

}

private function chargementDemarre ( pEvt:Event ):void

{

}

private function chargementEnCours ( pEvt:ProgressEvent ):void

{

}

private function chargementTermine ( pEvt:Event ):void
```

```
        {  
            // centre la visionneuse  
            visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) /  
2);  
            visionneuse.y = int ( ( stage.stageHeight - visionneuse.height )  
/ 2);  
        }  
  
        private function clicPrecedent ( pEvt:MouseEvent=null ):void  
        {  
            position = Math.max ( 0, --position );  
            requete.url = tableauImages [ position ];  
            visionneuse.load ( requete );  
        }  
  
        private function clicSuivant ( pEvt:MouseEvent=null ):void  
        {  
            position = ++position % tableauImages.length;  
            requete.url = tableauImages [ position ];  
            visionneuse.load ( requete );  
        }  
    }  
}
```

Lorsque le chargement démarre, nous ajoutons la barre de préchargement à l’affichage si celle-ci n’est pas déjà affichée :

```
private function chargementDemarre ( pEvt:Event ):void  
{  
    if ( !contains ( prechargeur ) ) addChild ( prechargeur );  
}
```

Puis nous adaptons sa taille selon le pourcentage chargé :

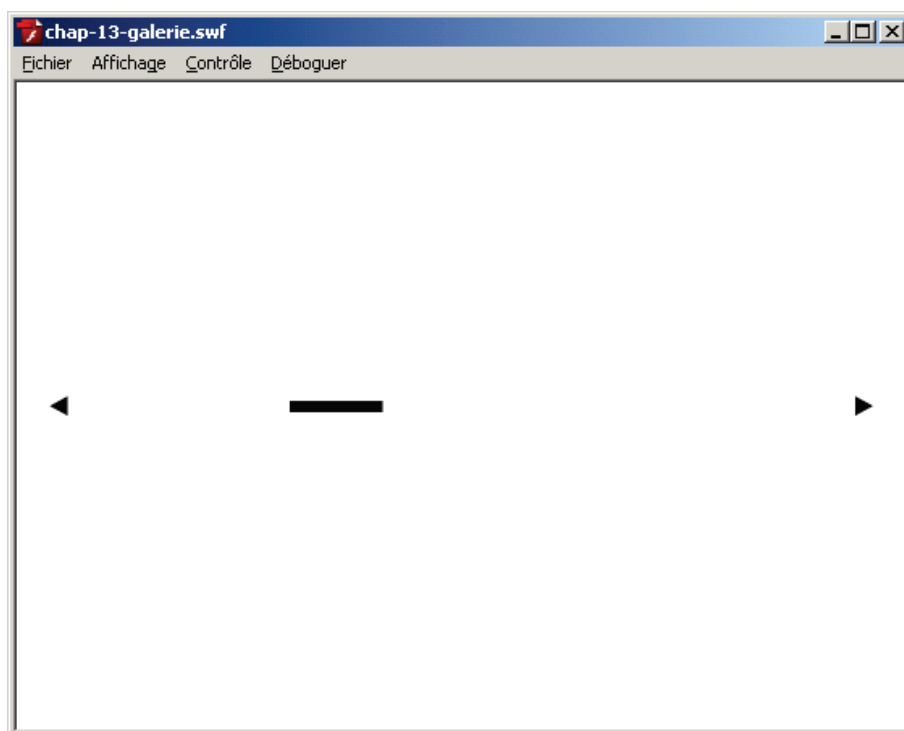
```
private function chargementEnCours ( pEvt:ProgressEvent ):void  
{  
    prechargeur.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;  
}
```

Enfin, nous supprimons la barre de préchargement lorsque les données sont chargées :

```
private function chargementTermine ( pEvt:Event ):void
{
    // centre la visionneuse
    visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) / 2);
    visionneuse.y = int ( ( stage.stageHeight - visionneuse.height ) / 2);

    if ( contains ( prechargeur ) ) removeChild ( prechargeur );
}
```

En testant notre galerie, chaque image est préchargée, comme l'illustre la figure 13-11 :



*Figure 13-11. Préchargement des images.*

Libre à vous d'intégrer différents types d'éléments indiquant le niveau de chargement des données. Nous aurions pu utiliser une barre de préchargement accompagnée d'un champ texte affichant un pourcentage de 0 à 100, ou bien une animation se jouant selon le volume de données chargées.

Afin d'ajouter un peu d'esthétisme à notre galerie, nous ajoutons un effet de fondu permettant aux photos d'être affichées progressivement :

```
package org.bytearray.document
{
    import flash.display.SimpleButton;
```

```
import flash.display.Sprite;
import flash.events.MouseEvent;
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.net.URLRequest;
import fl.transitions.Tween;
import fl.transitions.easing.Strong;
import org.bytearray.abstrait.ApplicationDefault;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefault
{
    // interface
    public var boutonSuivant:SimpleButton;
    public var boutonPrecedent:SimpleButton;
    public var prechargeur:Prechargeur;

    // visionneuse
    private var visionneuse:ChargeurMedia;

    // données
    private var tableauImages:Array;
    private var position:int;
    private var requete:URLRequest;

    // fondu
    private var fondu:Tween;

    public function Document ()
    {
        position = -1;

        requete = new URLRequest();

        // tableau contenant les url des images
        tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

        // création de l'objet Loader
        visionneuse = new ChargeurMedia ( 445, 335 );

        addChild ( visionneuse );

        fondu = new Tween ( visionneuse, "alpha", Strong.easeOut, 1, 1,
0, true );

        prechargeur = new Prechargeur();

        prechargeur.x = int ( (stage.stageWidth - prechargeur.width) /
2);
        prechargeur.y = int ( (stage.stageHeight - prechargeur.height) /
2);

        visionneuse.addEventListener ( Event.OPEN, chargementDemarre );
        visionneuse.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
        visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
);
    }
}
```

```

        boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
        boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

        clicSuivant ( new MouseEvent ( MouseEvent.CLICK ) );

    }

    private function chargementDemarre ( pEvt:Event ):void
    {
        if ( !contains ( prechargeur ) ) addChild ( prechargeur );
    }

    private function chargementEnCours ( pEvt:ProgressEvent ):void
    {
        prechargeur.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;
    }

    private function chargementTermine ( pEvt:Event ):void
    {
        // centre la visionneuse
        visionneuse.x = Math.round ( ( stage.stageWidth -
visionneuse.width ) / 2 );
        visionneuse.y = Math.round ( ( stage.stageHeight -
visionneuse.height ) / 2 );

        if ( contains ( prechargeur ) ) removeChild ( prechargeur );

        fondue.continueTo ( 1, 1 );
    }

    private function clicPrecedent ( pEvt:MouseEvent=null ):void
    {
        fondue.continueTo ( 0, 1 );

        position = Math.max ( 0, --position );

        requete.url = tableauImages [ position ];

        visionneuse.load ( requete );
    }

    private function clicSuivant ( pEvt:MouseEvent=null ):void
    {
        fondue.continueTo ( 0, 1 );

        position = ++position % tableauImages.length;
    }

```



```
        requete.url = tableauImages [ position ];  
        visionneuse.load ( requete );  
    }  
}  
}
```

En testant notre galerie, nous remarquons un effet de fondu entre chaque image. Nous pouvons rendre le type de transition dynamique en passant en paramètre un type de comportement spécifique.

A vous d’imaginer la suite, pour cela voici un indice :

En privilégiant une approche par composition, une classe comportementale pourrait être définie puis passée en paramètre à la classe `ChargeurMedia` afin de terminer un type de transition.

### A retenir

- Le contenu doit toujours être préchargé afin d’offrir une expérience utilisateur optimale.

## Interrompre le chargement

Il était impossible avec les précédentes versions d’ActionScript d’interrompre le chargement d’un élément externe. Le seul moyen était de fermer le lecteur Flash. ActionScript 3 intègre dorénavant une méthode `close` sur la classe `Loader` permettant de stopper instantanément le chargement d’un élément.

Afin de rendre notre classe `ChargeurMedia` souple, nous pouvons ajouter une nouvelle méthode `close` :

```
public function close ( ):void  
{  
    chargeur.close();  
}
```

Celle-ci délègue cette fonctionnalité à la classe `Loader` interne. Lorsque la méthode `load` est appelée, tout chargement précédemment initié est interrompu.

Nous n’avons pas abordé jusqu’à présent la notion de communication entre l’objet `Loader` et le contenu. Dans la partie suivante nous allons nous attarder sur la communication entre deux animations.

## A retenir

- La méthode `close` de l'objet `Loader` permet d'interrompre le chargement en cours.

## Communiquer entre deux animations

Dans un nouveau document Flash CS3 nous plaçons sur la scène une instance de symbole animé. L'animation est représentée par une instance du symbole `Garcon` utilisé lors du chapitre précédent. Nous lui donnons `animation` comme nom d'occurrence :

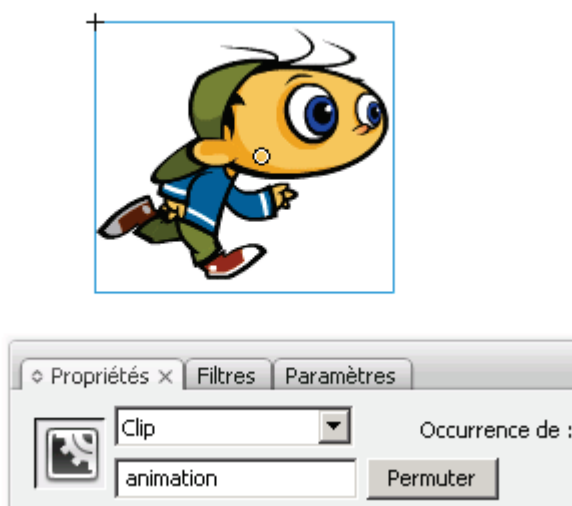


Figure 13-12. Instance du symbole `Garçon`.

Une fois l'animation exportée, nous la chargeons à l'aide de l'objet `Loader` depuis un autre SWF :

```
var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine );

chargeur.load ( new URLRequest ( "chap-13-anim.swf" ) );

addChild ( chargeur );

function termine ( pEvt:Event ):void
{
    // référence le scénario de l'animation chargée
    var scenario:DisplayObject = pEvt.target.content;

    // affiche : [object MainTimeline]
    trace( scenario );
}
```

```
}
```

Une fois l’animation chargée, nous pouvons stopper l’animation en utilisant la syntaxe pointée :

```
function termine ( pEvt:Event ):void
{
    // référence le scénario de l'animation chargée
    var scenario:DisplayObject = pEvt.target.content;

    // si le scénario est un MovieClip nous accédons
    // à l'animation et la stoppons
    if ( scenario is MovieClip ) MovieClip ( scenario ).animation.stop();
}
```

Si l’accès au contenu de l’animation chargée s’avère aisée, la communication inverse s’avère plus complexe.

Afin d’accéder au scénario du SWF initiant le chargement, nous utilisons la propriété `root` du scénario principal du SWF chargé, puis à la propriété `parent` de ce même scénario :

```
// affiche : [object Loader]
trace( root.parent );
```

Nous accédons ainsi à l’objet `Loader` chargeant actuellement notre animation. En ciblant la propriété `root` de ce même objet nous accédons au scénario principal du SWF chargeur :

```
// affiche : [object MainTimeline]
trace( root.parent.root );
```

Afin de cibler une animation posée sur ce même scénario, nous pouvons écrire le code suivant :

```
// référence le scénario principal du SWF parent
var scenarioPrincipal:DisplayObject = root.parent.root;

// si celui-ci est un MovieClip
if ( scenarioPrincipal is MovieClip )
{
    // alors nous transtypons et accédons au clip animation
    MovieClip ( scenarioPrincipal ).animation.stop();
}
```

Notons que si l’objet `Loader` initiant le chargement n’est pas présent au sein de la liste d’affichage, sa propriété `root` renverra `null`.

Dans l’exemple précédent, nous n’avons rencontré aucune difficulté à accéder au contenu l’animation chargée car les deux animations évoluent depuis le même domaine.

Une autre technique plus élégante consiste à diffuser un événement personnalisé depuis le SWF initiant le chargement. Le SWF chargé est à l'écoute de ce dernier et reçoit les informations de manière souple et élégante.

Pour réaliser cet échange, nous utilisons l'objet `EventDispatcher` disponible par la propriété `sharedEvents` de la classe `LoaderInfo`.

Au sein du SWF initiant le chargement, nous diffusons un événement personnalisé contenant les informations à transmettre :

```
var chargeur:Loader = new Loader();
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine );
chargeur.load ( new URLRequest ( "chap-13-anim.swf" ) );
addChild ( chargeur );
function termine ( pEvt:Event ):void
{
    // nous diffusons un événement EvenementInfos.INFOF au SWF chargé
    pEvt.target.sharedEvents.dispatchEvent ( new EvenementInfos (
    EvenementInfos.INFOF, "contenu !" ) );
}
```

Au sein du SWF chargé nous écoutons ce même événement :

```
loaderInfo.sharedEvents.addEventListener ( EvenementInfos.INFOF, ecouteur );
function ecouteur ( pEvt:EvenementInfos ):void
{
    // affiche : contenu !
    trace( pEvt.infos );
}
```

Cette approche facilite grandement la communication entre deux animations et doit être considérée en priorité car elle ne souffre d'aucune restriction de sécurité même en contexte interdomaine.

Dans un contexte de chargement de contenu externe, nous risquons d'être confrontés aux restrictions de sécurité du lecteur Flash. Dans la partie suivante nous allons faire le point sur ces limitations afin de mieux comprendre comment les appréhender et résoudre certains problèmes.

## Modèle de sécurité du lecteur Flash

Depuis le lecteur Flash 6, des restrictions de sécurité ont été ajoutées au sein du lecteur Flash lors du chargement ou d'envoi de données afin de protéger les auteurs de contenu divers.

Le modèle de sécurité du lecteur Flash appelé *Security Sandbox* en Anglais distingue deux acteurs :

- Le créateur du contenu
- Le chargeur de contenu

Ce modèle s'applique à tout type de contenu chargé au sein du lecteur. Il faut comprendre que par principe, lorsqu'un contenu graphique est chargé depuis un autre domaine, celui-ci est en *lecture seule* au sein du lecteur. Ce dernier refuse de scripter ou de modifier tout contenu graphique provenant d'un autre domaine. On dit alors que les fichiers évoluent dans un contexte *interdomaine*.

Par le terme *scripter* nous entendons l'accès par ActionScript à des données contenues dans une autre animation. Adobe utilise le terme de *programmation croisée* pour exprimer ce mécanisme.

Par le terme de *modification*, nous entendons par exemple l'activation de la propriété *smoothing* sur une image bitmap chargée, ou encore la capture d'une vidéo sous forme bitmap à l'aide de la méthode *draw* de la classe *BitmapData*.

Afin de bien comprendre le modèle de sécurité, nous allons étudier différentes situations. Commençons par le cas de figure suivant :

Nous développons un portail de jeux Flash censé charger des jeux provenant de différents sites. Si le lecteur Flash n'intégrait pas de modèle de sécurité, il serait possible d'ajouter du code aux différents jeux chargés et de pirater ces derniers.

Si toutefois, nous devons accéder au code d'un jeu chargé, nous devons ajouter au sein de celui-ci une ligne de code autorisant le domaine spécifique à scripter l'animation. En d'autres termes, le contenu chargé doit autoriser le chargeur à le scripter.

Dans le cas de chargement d'images provenant d'autres domaines, celles-ci n'ont pas la possibilité d'autoriser le chargeur par ActionScript. Nous utilisons dans cas des fichiers XML contenant la liste des domaines autorisés. Ces fichiers sont appelés *fichiers de régulation*.

---

### A retenir

---

- Par défaut, le lecteur Flash empêche de scripter ou modifier tout contenu graphique provenant d'un domaine différent.
- En revanche, la lecture seule d'animation ou images est toujours autorisée quelque soit le contexte.
- Le chargement de données type XML ou autres est toujours interdit dans un contexte interdomaine.

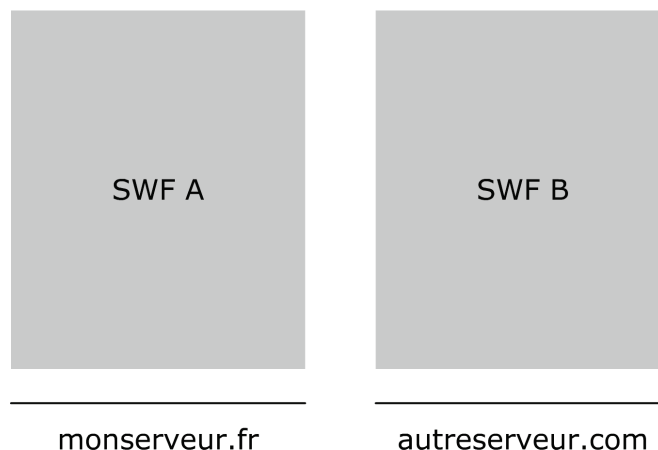
## Programmation croisée

Nous entendons par le terme de *programmation croisée*, l'accès par ActionScript au code contenu par un SWF.

Nous avons vu au cours de la précédente partie comment faire communiquer deux animations situées sur un même domaine. Nous savons que par défaut, la communication est toujours autorisée entre deux SWF résidant sur le même domaine.

A l'inverse, lorsque deux animations souhaitent communiquer mais ne résident pas sur le même domaine, l'animation devant être scriptée doit autoriser l'animation ayant amorcé le chargement.

La figure 13-13 illustre la situation :



*Figure 13-13. Animations en contexte interdomaine.*

Dans le schéma illustré par la figure 13-15 nous voyons deux animations en contexte *interdomaine*.

L'animation `SWF A` souhaite scripter l'animation `SWF B`. Pour cela, cette dernière doit appeler la méthode `allowDomain` de la classe `flash.system.Security` dont voici la signature :

```
public static function allowDomain(... domains):void
```

Nous pouvons passer en paramètre les domaines autorisés à scripter le SWF en cours. Ainsi nous placerons au sein de l'animation SWF le code suivant :

```
Security.allowDomain("monserveur.fr");
```

Il n'est pas nécessaire d'ajouter http devant le domaine, nous ne spécifions généralement que le nom et le domaine. Il est aussi possible de spécifier une adresse IP.

## A retenir

- Dans un contexte interdomaine, l'accès par ActionScript au code contenu par un SWF est appelé *programmation croisée*.

## Utiliser un fichier de régulation

Comme expliqué précédemment, dans le cas de chargement d'images bitmap provenant d'un serveur distant, il est impossible d'ajouter au sein de celles-ci un appel à la méthode `allowDomain`. Afin de pallier à ce problème, nous pouvons créer des fichiers de régulation.

Ces derniers doivent être placés sur le serveur hébergeant les images et sont en réalité de simples fichiers XML sauvés sous le nom de `crossdomain.xml`.

La figure 13-14 illustre l'idée :

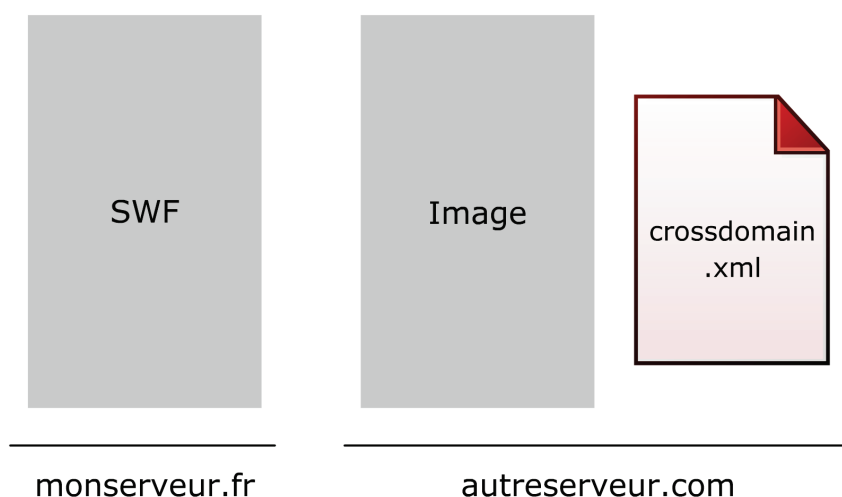


Figure 13-14. Fichier de régulation.

Lorsque le lecteur Flash charge une image, celui-ci tentait dans ses précédentes versions de charger automatiquement le fichier de régulation depuis la racine du serveur.

Depuis le lecteur 9, il est nécessaire d'indiquer s'il est nécessaire de charger le fichier de régulation avant de commencer le chargement de l'élément. L'utilisation de fichiers de régulation est fondamentale pour les sites hébergeant des images utilisées au sein d'applications Flash.

Voici une liste de sites utilisant un fichier de régulation :

- <http://www.facebook.com/crossdomain.xml>
- <http://www.adobe.com/crossdomain.xml>
- <http://www.youtube.com/crossdomain.xml>
- <http://static.flickr.com/crossdomain.xml>

En analysant le contenu d'un fichier de régulation nous découvrons un simple fichier XML contenant la liste des domaines autorisés à modifier les images. Ces derniers sont appelés plus couramment *domaines de confiance*.

Voici le contenu du fichier de régulation situé sur le serveur *YouTube* :

```
<cross-domain-policy>
<allow-access-from domain="*.youtube.com"/>
<allow-access-from domain="*.google.com"/>
</cross-domain-policy>
```

Ce fichier de régulation indique que seuls les SWF hébergés dans des sous-domaines de youtube.com ou google.com peuvent scripter les images hébergées sur youtube.com.

En analysant le fichier de régulation de flickr, nous remarquons que ces derniers sont beaucoup plus permissifs :

```
<cross-domain-policy>
<allow-access-from domain="*"/>
</cross-domain-policy>
```

Nous découvrons que tous les domaines peuvent modifier les images hébergées sur flickr.com.

---

## A retenir

---



- Dans le cas de chargement d'images distantes, des fichiers de régulation peuvent être créés afin d'autoriser les domaines de confiance.
- Un fichier de régulation est un simple fichier XML.
- Celui-ci doit être nommé par défaut `crossdomain.xml`.

## Contexte de chargement

Lorsque les méthodes `load` et `loadBytes` de la classe `Loader` sont appelées, nous pouvons passer en deuxième paramètre un contexte de chargement exprimé par la classe `flash.system.LoaderContext`.

La classe `LoaderContext` permet d'indiquer le domaine d'application et de sécurité dans lequel sera placé le contenu.

Le constructeur de la classe `LoaderContext` accepte trois paramètres dont voici le détail :

- `checkPolicyFile` : indique si un fichier de régulation doit être chargé avant de commencer à charger le contenu.
- `applicationDomain` : sert à préciser le domaine d'application à utiliser une fois le contenu chargé. La notion de domaine d'application est traitée dans la partie intitulée *Bibliothèque partagée*.
- `securityDomain` : représente le modèle de sécurité. Il est seulement utilisé lors du chargement de fichiers SWF. Son rôle est traité dans la partie intitulée *Bibliothèque partagée*.

Ainsi, afin de pouvoir modifier une image hébergée depuis un domaine distant nous demandons au lecteur Flash de charger un fichier de régulation :

```
var chargeur:Loader = new Loader();

var requete:URLRequest = new URLRequest
("http://www.serveurdistant.com/images/wiiflash.jpg");

var contexte:LoaderContext = new LoaderContext ( true );

chargeur.load ( requete, contexte );
```

Automatiquement, le lecteur Flash tente de charger un fichier de régulation stocké à la racine du serveur `serveurdistant`. Si un tel fichier nommé `crossdomain.xml` est présent et autorise notre domaine alors nous pouvons modifier l'image chargée.

Si aucun fichier de régulation n'est trouvé, le lecteur Flash empêche toute modification de l'image chargée.

Pour des questions de pratique, il vous est peut-être impossible de placer un fichier de régulation à la racine du domaine distant. Si le

fichier de régulation est placé à un emplacement différent de la racine du domaine distant. Nous spécifions son chemin à l'aide de la méthode `loadPolicyFile` de la classe `Security`.

Dans le code suivant, nous spécifions au lecteur de ne pas chercher le fichier de régulation à la racine du domaine mais au sein du répertoire

`images` :

```
var chargeur:Loader = new Loader();  
  
Security.loadPolicyFile("http://serveurdistant.com/images/regulation.xml");  
  
var requete:URLRequest = new URLRequest  
("http://www.serveurdistant.com/images/wiiflash.jpg");  
  
var contexte:LoaderContext = new LoaderContext ( true );  
  
chargeur.load ( requete, contexte );
```

Notons que grâce à la méthode `loadPolicyFile` nous pouvons aussi spécifier le nom du fichier de régulation, ici `regulation.xml`.

Attention, l'emplacement du fichier de régulation a une importance. A la racine, celui-ci autorise l'accès à tous les fichiers du site. S'il se trouve plus loin dans l'arborescence, il n'autorisera que les fichiers de ce dossier ainsi que ceux des dossiers enfants.

## Contourner les restrictions de sécurité

Nous avons vu dans la partie intitulée *Programmation croisée* qu'il était possible d'autoriser la manipulation d'images hébergées sur des domaines distants par la création de fichiers de régulation.

Malheureusement, dans certains cas, l'ajout de tels fichiers est impossible. Certains sites prévoient quelquefois leur installation, mais ils demeurent minoritaires. Il peut donc être intéressant de savoir contourner les restrictions de sécurité du lecteur Flash. C'est ce que nous allons apprendre dès maintenant.

Imaginons le cas suivant :

Vous venez d'apprendre que vous devez développer une application Flash basée sur des images provenant de différentes sources. En d'autres termes, chaque image devra être chargée tout en étant hébergée sur n'importe quel domaine.

Pour l'instant, cela ne pose aucun problème car le simple chargement d'images en situation *interdomaine* est autorisé. En cours de développement vous vous rendez compte qu'un lissage est nécessaire et qu'il serait intéressant de pouvoir l'activer auprès des images chargées.

Un premier réflexe vous incite à activer la propriété `smoothing` sur l'objet `Bitmap` chargé :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine );

// image google map
var requete:URLRequest = new URLRequest (
    "http://kh0.google.fr/kh?n=404&v=22&t=trtqttqrrqrqssts" );

// chargement de l'image
chargeur.load ( requete );

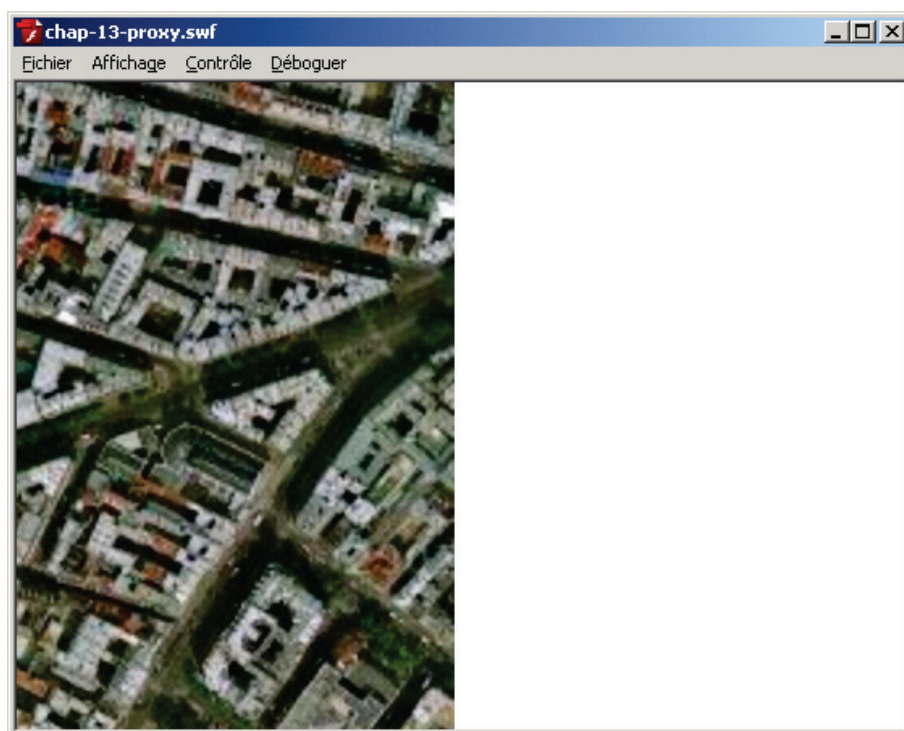
// ajout à la liste d'affichage
addChild ( chargeur );

function termine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // accès à l'image bitmap
    var image:Bitmap = Bitmap ( objetLoaderInfo.content );

    // activation du lissage
    image.smoothing = true;
}
```

En testant l'application en local, le lissage fonctionne, la figure 13-15 illustre le résultat :



*Figure 13-15. Image lissée provenant de Google Map.*

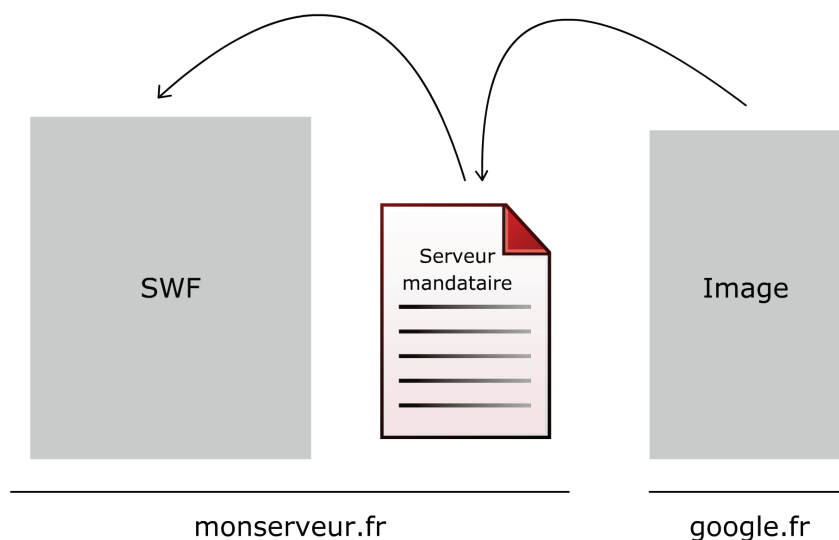
Malheureusement, une fois l'application publiée sur votre serveur tout accès au contenu chargé lève une erreur à l'exécution de type `SecurityError` :

```
SecurityError: Error #2122: Violation de la sécurité Sandbox :  
LoaderInfo.content : http://www.bytearray.org/pratique-as3/chap-13-proxy.swf  
ne peut pas accéder à http://kh0.google.fr/kh?n=404&v=22&t=trtqttqrrrqrqssts.  
Un fichier de régulation est nécessaire, mais l'indicateur checkPolicyFile  
n'a pas été défini lors du chargement de ce support.
```

Ceci est dû au fait que le serveur Google ne possède aucun fichier de régulation nous autorisant à manipuler ses images. L'utilisation de la propriété `checkPolicyFile` est donc dans ce cas précis inutile. En d'autres termes, le contenu chargé est en lecture seule.

Nous allons donc utiliser une astuce consistant à faire croire au lecteur Flash que nous chargeons un élément provenant du même domaine. Pour cela, nous utilisons un relais, plus couramment appelé *serveur mandataire*.

La figure 13-16 illustre le concept :



*Figure 13-16. Chargement d'images par serveur mandataire.*

L'astuce consiste à charger l'image depuis le serveur mandataire, qui est ensuite chargé par le lecteur Flash. Ce dernier pense charger un élément provenant du même domaine, sans penser que le serveur mandataire contient l'image provenant du domaine distant.

La création du serveur mandataire se limite à deux lignes de code PHP. Nous utilisons dans notre exemple le langage serveur PHP qui s'avère être un des langages les plus efficace pour travailler avec Flash.

Au sein d'un fichier intitulé `proxy.php` nous ajoutons le script suivant :

```
<?php

$chemin = $_POST["chemin"];
readfile($chemin);

?>
```

Nous passons par le tableau `$_POST` le chemin d'image. Puis la fonction PHP `readfile` renvoie le flux d'image directement au lecteur Flash.

Puis nous modifions le code, afin de charger l'image par le serveur mandataire :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine);
```

```
// image google map
var requete:URLRequest = new URLRequest ( "proxy.php" );

// création d'un objet URLVariables permettant
// de passer des variables au serveur mandataire
var variables:URLVariables = new URLVariables();

// création de la variable chemin
variables.chemin = "http://kh0.google.fr/kh?n=404&v=22&t=trtqttqqrqrqssts";

// les variables doivent être passées par la requete HTTP
requete.data = variables;

// les variables sont envoyées au sein du tableau POST
requete.method = URLRequestMethod.POST;

// chargement de l'image
chargeur.load ( requete );

// ajout à la liste d'affichage
addChild ( chargeur );

function termine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // accès à l'image bitmap chargée
    var image:Bitmap = Bitmap ( objetLoaderInfo.content );
}
```

Nous utilisons la classe `URLVariables` afin de passer l'adresse de l'image à charger au serveur mandataire. Nous reviendrons en détail sur cette classe au cours du chapitre 14 intitulé *Chargement et envoi de données*.

Une fois publiée, si nous lançons l'application, celle-ci ne lève plus d'erreur à l'exécution lorsque nous accédons à la propriété `content`.

Nous pouvons ainsi activer le lissage en passant la valeur booléenne `true` à la propriété `smoothing` de l'objet `Bitmap` chargé :

```
function termine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // accès à l'image bitmap chargée
    var image:Bitmap = Bitmap ( objetLoaderInfo.content );

    // activation du lissage
    image.smoothing = true;
}
```

La figure 13-17 illustre la différence entre l'image lissée et non lissée :



*Figure 13-17. Image non lissée et lissée.*

De la même manière, il peut être nécessaire de rendre sous forme bitmap un objet Loader contenant une image provenant d'un domaine distant :

```
function termine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // création d'une instance de BitmapData
    var donneesBitmap:BitmapData = new BitmapData ( objetLoaderInfo.width,
    objetLoaderInfo.height );

    // l'objet Loader est rendu sous forme bitmap
    donneesBitmap.draw ( pEvt.target.loader );

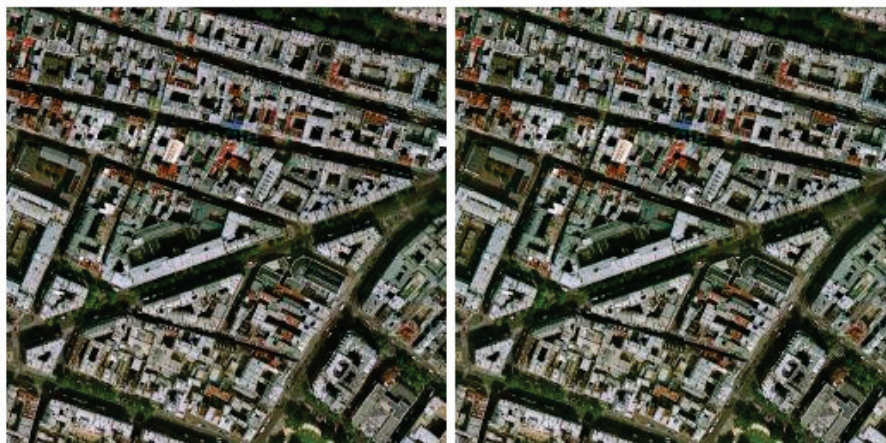
    var image:Bitmap = new Bitmap ( donneesBitmap );

    image.x = objetLoaderInfo.width + 5;

    addChild ( image );
}
```

Ce qui génère le résultat illustré en figure 13-18 :





*Figure 13-18. Image dupliquée.*

Sans serveur mandataire, l'appel de la méthode `draw` aurait levé une erreur de sécurité à l'exécution.

Cette technique de serveur mandataire est une solution efficace qui possède malheureusement un inconvénient. Au lieu d'être directement chargée depuis le client, l'image est d'abord chargée par le serveur puis chargée par le client. La charge serveur peut donc être plus importante et à surveiller sur un grand projet destiné à un trafic important.

### A retenir

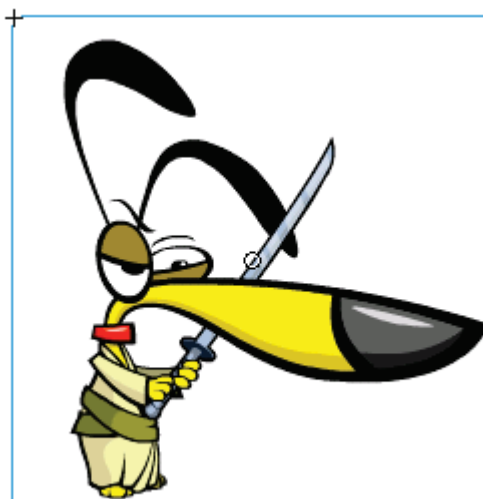
- L'utilisation d'un serveur mandataire permet de charger et modifier tout type de contenu provenant d'un différent domaine.
- C'est une solution simple et efficace mais qui peut entraîner une charge serveur plus importante.

## Bibliothèque partagée

Dans le cas de chargement d'animations, il peut être parfois utile d'extraire une classe utilisée au sein d'un SWF afin de l'utiliser au sein de l'animation procédant au chargement. Cela est rendu possible grâce au mécanisme de *bibliothèque partagée à l'exécution* apporté par la classe `flash.system.ApplicationDomain`.

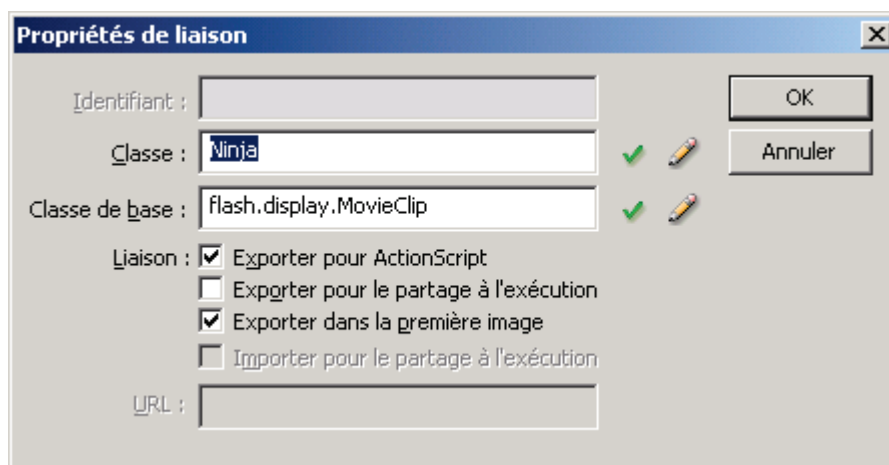
Dans un nouveau document Flash nous créons un symbole clip comme illustré en figure 13-19 :





*Figure 13-19. Symbole clip.*

Puis nous lions le symbole à une classe `Ninja` grâce au panneau de *Propriétés de liaisons* :



*Figure 13-20. Panneau de propriétés de liaison.*

Une fois définie, nous exportons simplement l’animation sous le nom de `bibliotheque.swf`. Nous allons maintenant charger ce fichier SWF et accéder dynamiquement à la classe `Ninja`.

Dans un nouveau document Flash, nous créons un objet `Loader` puis nous chargeons l’animation `bibliotheque.swf` :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
```

```
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargeurTerminé );

// chargement de l'animation contenant la classe Ninja
chargeur.load ( new URLRequest ("bibliotheque.swf") );

function chargeurTerminé ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // affiche : [object LoaderInfo]
    trace( objetLoaderInfo );
}
```

Une fois l'animation chargée, nous pouvons accéder à toutes les classes définies au sein de celle-ci grâce à la méthode `getDefinition` de la classe. Celle-ci peut être appelée dès lors que l'événement `Event.INIT` est diffusé. Nous n'ajoutons pas volontairement l'objet `Loader` à la liste d'affichage car nous souhaitons simplement extraire une classe partagée.

Souvenez-vous, nous avons vu précédemment que la classe `LoaderInfo` possède une propriété `applicationDomain` utilisée lors du chargement de SWF.

Celle-ci référence un objet appelé *Domaine d'application* :

```
function chargeurTerminé ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // référence le domaine d'application du SWF chargé
    var domaineApplication:ApplicationDomain =
    objetLoaderInfo.applicationDomain;

    // affiche : [object ApplicationDomain]
    trace( domaineApplication );
}
```

Le domaine d'application est un objet dans lequel sont placés toutes les définitions de classe d'un SWF. Ainsi, le domaine d'application de l'animation `bibliotheque.swf` contient une classe `Ninja`.

La classe `ApplicationDomain` possède deux méthodes dont voici le détail :

- `getDefinition` : Extrait une définition de classe spécifique.
- `hasDefinition` : Indique si la définition de classe existe.

Deux propriétés peuvent aussi être utilisées :

- `currentDomain` : Référence le domaine d'application du SWF en cours.
- `parentDomain` : Référence le domaine d'application parent.

Nous allons extraire la classe `Ninja` du domaine d'application du SWF chargé puis l'instancier et afficher le symbole au sein de l'animation procédant au chargement :

```
function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // référence le domaine d'application du SWF chargé
    var domaineApplication:ApplicationDomain =
    objetLoaderInfo.applicationDomain;

    // extrait la définition de classe Ninja
    var importNinja:Class = Class ( domaineApplication.getDefinition( "Ninja" )
    );

    // création d'une instance de Ninja
    var instanceNinja:DisplayObject = new importNinja();

    // ajout à la liste d'affichage
    addChild ( instanceNinja );
}
```

La figure 13-21 illustre le résultat :



*Figure 13-21. Affichage du symbole **Ninja**.*

Si nous tentons d'extraire une classe inexistante :

```
// tente d'extraire une définition classe nommée Nina
var importNinja:Class = Class ( domaineApplication.getDefinition( "Nina" ) );
```

Une erreur de type **ReferenceError** est levée :

```
ReferenceError: Error #1065: La variable Nina n'est pas définie.
```

A l'aide d'un bloc, **try catch** nous pouvons gérer l'erreur et ainsi réagir :

```
function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // référence le domaine d'application du SWF chargé
    var domaineApplication:ApplicationDomain =
    objetLoaderInfo.applicationDomain;

    try
    {

        // tentative d' extraction de la définition de classe
        var importNinja:Class = Class ( domaineApplication.getDefinition(
        "Nina" ) );

        // création d'une instance de Ninja
        var instanceNinja:DisplayObject = new importNinja();

        // ajout à la liste d'affichage
        addChild ( instanceNinja );

    } catch ( pError:Error )
    {

        trace("La définition de classe spécifiée n'est pas disponible");

    }
}
```

Si l'utilisation d'un bloc **try catch** ne vous convient pas, l'appel de la méthode **hasDefinition** offre un résultat équivalent :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

// chargement de l'animation contenant la classe Ninja
chargeur.load ( new URLRequest ( "librairie.swf" ) );
```

```
var definitionClasse:String = "Ninja";

function chargementTermine ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // référence le domaine d'application du SWF chargé
    var domaineApplication:ApplicationDomain =
    objetLoaderInfo.applicationDomain;

    // vérifie si la définition de classe Ninja est disponible
    if ( domaineApplication.hasDefinition( definitionClasse ) )
    {
        // tentative d' extraction de la définition de classe
        var importNinja:Class = Class ( domaineApplication.getDefinition(
        definitionClasse ) );

        // création d'une instance de Ninja
        var instanceNinja:DisplayObject = new importNinja();

        // ajout à la liste d'affichage
        addChild ( instanceNinja );

        } else trace ("La définition de classe " + definitionClasse + " n'est pas
        disponible");
    }
}
```

L'extraction de classes est rendue possible car l'animation contenant les classes à extraire provient du même domaine que l'animation procédant au chargement.

Bien entendu, le modèle de sécurité du lecteur Flash empêche par défaut l'extraction de classes entre deux SWF évoluant dans un contexte interdomaine. Dans ce cas, nous devons explicitement demander au lecteur Flash de placer le SWF chargé dans le même domaine de sécurité afin de pouvoir extraire les classes.

Une première approche consiste à appeler la méthode `allowDomain` de la classe `Security` depuis le SWF dont les classes sont extraites.

```
Security.allowDomain("monserveurDeConfiance.fr");
```

La seconde requiert le placement d'un fichier de régulation sur le domaine du SWF à charger, puis de passer un objet `LoaderContext` à la méthode `load` de l'objet `Loader` en spécifiant le domaine de sécurité en cours par la propriété `securityDomain` :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );
```

```
// création d'un objet de contexte
var contexte:LoaderContext = new LoaderContext();

// nous demandons de placer le SWF chargé dans le même domaine
// de sécurité afin de pouvoir extraire ses classes
contexte.securityDomain = SecurityDomain.currentDomain;

// chargement de l'animation contenant la classe Ninja
// en spécifiant le contexte
chargeur.load ( new URLRequest
("http://serveurdistant.com/swf/bibliotheque.swf"), contexte );

function chargementTermine ( pEvt:Event ):void
{

    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // référence le domaine d'application du SWF chargé
    var domaineApplication:ApplicationDomain =
    objetLoaderInfo.applicationDomain;

    // extrait la définition de classe Ninja
    var importNinja:Class = Class ( domaineApplication.getDefinition( "Ninja" )
);

    // création d'une instance de Ninja
    var instanceNinja:DisplayObject = new importNinja();

    // ajout à la liste d'affichage
    addChild ( instanceNinja );

}
```

Si pour des questions de pratique, vous n’avez pas la possibilité d’appeler la méthode `allowDomain` de la classe `Security` ou de placer un fichier de régulation sur le domaine distant, l’utilisation d’un fichier serveur mandataire est ici aussi envisageable.

Grâce à ce concept d’import dynamique de classes, nous pouvons imaginer toutes sortes d’applications tirant profit d’une telle fonctionnalité. Une application Flash pourrait importer, dès son initialisation un SWF contenant les définitions de classe nécessaires. Celles-ci seraient dynamiquement instanciées puis utilisées dans l’application.

L’application reposerait donc entièrement sur ces classes importées dynamiquement. Afin de mettre à jour l’application, nous pourrions simplement régénérer le SWF contenant les définitions de classe.

L’application pourrait être mise à jour de la même manière que des `.dll` dans d’autres langages comme C++ ou C#.

---

## A retenir

---

- La méthode `getDefinition` de la classe `ApplicationDomain` permet d'extraire une définition de classe contenue dans un SWF.
- Cette extraction est soumise au modèle de sécurité du lecteur Flash.
- Afin de pouvoir extraire une classe d'un SWF distant, celui-ci doit autoriser l'animation ayant amorcé le chargement par la méthode `allowDomain` de la classe `Security` ou la création d'un fichier de régulation.

## Désactiver une animation chargée

Très souvent, un site Flash est constitué d'une application principale chargeant différents modules, séparés en plusieurs SWF. Chacun d'entre eux est ensuite chargé afin de naviguer dans le site.

Le fonctionnement de la classe `Loader` nous réserve encore quelques surprises. En réalité, la méthode `unload` vide le contenu chargé mais ne le désactive pas. Cela diffère du traditionnel `loadMovie` utilisé en ActionScript 1 et 2, qui remplaçait le contenu précédemment chargé en désactivant tous les objets contenus.

En ActionScript 3, lorsque la méthode `unload` est exécutée, le contenu chargé est simplement supprimé de la liste d'affichage. La seule référence à l'animation est celle que possède l'objet `Loader`. Si nous supprimons son contenu il n'existe alors plus aucune référence vers l'animation. Celle-ci va donc demeurer et vivre en mémoire jusqu'à ce que le ramasse miettes intervienne et la supprime définitivement.

Ainsi, au chargement d'une nouvelle rubrique, le son de la précédente continuerait de jouer. De la même manière, tous les événements souscrits continueraient d'être diffusés.

Il faut donc prévoir *obligatoirement* un mécanisme de désactivation comme nous l'avons fait jusqu'à présent pour les objets d'affichage.

Afin de correctement désactiver une animation, nous utilisons l'événement `Event.UNLOAD` diffusé par l'objet `LoaderInfo` associé au SWF en cours. Le code suivant doit donc être placé au sein du SWF à désactiver :

```
// écoute la suppression de l'animation
loaderInfo.addEventListener ( Event.UNLOAD, desactivation );

function desactivation ( pEvt:Event ):void
{
    // logique de désactivation de l'animation
    // désactivation des événements, du son, objets videos etc
```

```
}  
}
```

Lorsque la méthode `unload` est exécutée, ou qu'une nouvelle animation est chargée, l'événement `Event.UNLOAD` est diffusé au sein de l'animation chargée. Nous intégrons au sein de la fonction écouteur `desactivation` la logique nécessaire afin de désactiver totalement l'animation en cours.

Dans le code suivant, nous stoppons le son en cours de lecture :

```
// création d'un objet Sound  
var monSon:Sound = new Sound();  
  
// chargement du son  
monSon.load ( new URLRequest ("son.mp3") );  
  
// création d'un objet SoundChannel par l'appel de la méthode Sound.play()  
var canalAudio:SoundChannel = monSon.play();  
  
// écoute la suppression de l'animation  
loaderInfo.addEventListener ( Event.UNLOAD, desactivation );  
  
function desactivation ( pEvt:Event ):void  
{  
  
    // logique de désactivation de l'animation  
    // désactivation des événements, du son, objets videos etc  
    canalAudio.stop();  
  
}
```

L'animation chargée étant supprimée de la liste d'affichage, l'écoute de l'événement `Event.REMOVED_FROM_STAGE` est aussi envisageable :

```
// création d'un objet Sound  
var monSon:Sound = new Sound ();  
  
// chargement du son  
monSon.load ( new URLRequest ("son.mp3") );  
  
// création d'un objet SoundChannel par l'appel de la méthode Sound.play()  
var canalAudio:SoundChannel = monSon.play();  
  
// écoute la suppression de l'animation  
addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );  
  
function desactivation ( pEvt:Event ):void  
{  
  
    // logique de désactivation de l'animation  
    // désactivation des événements, du son, objets videos etc  
    canalAudio.stop();  
  
}
```



Ce comportement peut poser de graves problèmes lorsque vous n’êtes pas l’auteur du contenu chargé. Vous êtes donc dans l’incapacité d’intégrer un mécanisme de désactivation. Il n’existe aujourd’hui aucune solution viable permettant de désactiver automatiquement un contenu tiers.

### A retenir

- Lorsque la méthode `unload` est appelée, le contenu est supprimé de la liste d’affichage mais n’est pas désactivé.
- Il est impératif de prévoir un mécanisme de désactivation au sein des animations chargées.
- Il n’est pas possible de désactiver automatiquement un contenu tiers.

## Communication AVM1 et AVM2

La machine virtuelle ActionScript 3 (AVM2) offre la possibilité de lire des animations développées en ActionScript 1 et 2 (AVM1). Celles-ci sont alors considérées comme des objets de type `flash.display.AVM1Movie`.

Dans le cas d’un portail de jeux vidéo développé en ActionScript 3, la majorité des jeux chargés seront d’ancienne génération, développés en ActionScript 1 ou 2. Malheureusement, l’échange entre les deux animations n’est pas simplifié.

Si nous tentons d’accéder au contenu de ces derniers, le lecteur lève une erreur indiquant que l’accès est impossible. Il faut considérer un objet `AVM1Movie` comme un objet hermétique ne pouvant être pénétré.

Afin de mettre en évidence ce comportement, nous allons créer une animation ActionScript 1 ou 2 et y intégrer une simple animation. L’animation est représentée par une instance du symbole `Garcon` utilisé lors du chapitre précédent. Nous lui donnons `animation` comme nom d’occurrence.



*Figure 13-22. Instance du symbole Garçon.*

Nous allons depuis l'animation ActionScript 3, communiquer avec le contenu l'animation d'ancienne génération pour stopper l'animation du clip `animation`.

Dans le code suivant, nous chargeons l'animation d'ancienne génération :

```
var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargeurTerminé );

chargeur.load ( new URLRequest ( "anim-vml.swf" ) );

addChild ( chargeur );

function chargeurTerminé ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

    // affiche : [object AVMLMovie]
    trace( objetLoaderInfo.content );
}
```

Nous remarquons que la propriété `content` de l'objet `LoaderInfo` nous renvoie un objet de type `AVMLMovie`. Si nous tentons de pénétrer à l'intérieur de l'animation :

```
function chargeurTerminé ( pEvt:Event ):void
{
    var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );
```

```
var contenu:DisplayObject = objetLoaderInfo.content;

if ( contenu is AVMLMovie )
{
    var animationVM1:AVMLMovie = AVMLMovie ( contenu );

    // tentative d'accès à l'occurrence animation
    trace( animationVM1.animation );
}
}
```

L'erreur à la compilation suivante est générée :

```
1119: Accès à la propriété animation peut-être non définie, via la référence
de type static flash.display:AVMLMovie.
```

Afin d'accéder au contenu de l'animation chargée, nous devons passer par un moyen détourné. Deux classes vont nous permettre de communiquer :

- `flash.net.LocalConnection` : la classe `LocalConnection` permet d'échanger des données entre différents SWF distincts.
- `flash.external.ExternalInterface` : la classe `ExternalInterface` permet la communication entre le code ActionScript et la page contenant le lecteur Flash.

Nous allons utiliser pour cet exemple la classe `LocalConnection` qui s'avère être la solution la plus souple, en ne nécessitant pas de code JavaScript contrairement à la classe `ExternalInterface`.

Nous commençons par créer une instance de la classe `LocalConnection` dans l'animation dans laquelle nous souhaitons accéder :

```
// création de l'objet récepteur
var recepateur:LocalConnection = new LocalConnection();

// connexion au canal utilisé par l'émetteur
recepateur.connect ( "canalCommunication" );

// définition de la méthode appelée par l'émetteur
recepateur.stopAnimation = function ( )

{
    animation.stop();
}
```

Puis au sein de l'animation souhaitant initier la communication, nous ajoutons une nouvelle instance de la classe `LocalConnection` afin d'émettre les messages :

```
var chargeur:Loader = new Loader();
chargeur.load ( new URLRequest ("anim-vm1.swf" ));
addChild ( chargeur );

// création de l'objet émetteur
var emetteur:LocalConnection = new LocalConnection();

boutonStop.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void
{
    // émission d'un message pour exécuter la méthode stopAnimation par le
    canal canalCommunication
    emetteur.send ("canalCommunication ", "stopAnimation");
}
```

Lorsque nous cliquons sur le bouton `boutonStop`, un message est envoyé à l'animation chargée par l'appel de la méthode `send` de l'objet `LocalConnection`.

### A retenir

- La communication entre deux animations AVM1 et AVM2 est possible par l'intermédiaire de la classe `LocalConnection` ou `ExternalInterface`.

Nous savons désormais comment charger du contenu graphique au sein du lecteur Flash. Passons maintenant au chargement et à l'envoi de données en ActionScript 3.

# 14

## Chargement et envoi de données

<b>LA CLASSE URLLOADER .....</b>	<b>1</b>
CHARGER DU TEXTE.....	3
L'ENCODAGE URL.....	13
CHARGER DES VARIABLES .....	14
CHARGER DES DONNÉES XML.....	20
<b>CHARGEMENT DE DONNEES ET SECURITE .....</b>	<b>30</b>
<b>CHARGER DES DONNÉES BINAIRES .....</b>	<b>32</b>
<b>CONCEPT D'ENVOI DE DONNEES .....</b>	<b>37</b>
ENVOYER DES VARIABLES.....	38
LA METHODE GET OU POST .....	45
ENVOYER DES VARIABLES DISCRÈTEMENT.....	48
RENNVOYER DES DONNEES DEPUIS LE SERVEUR.....	50
ALLER PLUS LOIN .....	56
ENVOYER UN FLUX BINAIRE.....	57
<b>TELECHARGER UN FICHIER.....</b>	<b>58</b>
PUBLIER UN FICHIER .....	68
PUBLIER PLUSIEURS FICHIERS .....	73
CREATION DE LA CLASSE ENVOI MULTIPLE.....	80
RETOURNER DES DONNEES UNE FOIS L'ENVOI TERMINE.....	89
<b>ALLER PLUS LOIN.....</b>	<b>93</b>

### La classe URLLoader

Afin de concevoir une application dynamique nous avons besoin de pouvoir envoyer ou charger des données externes au sein du lecteur Flash. Le chargement dynamique de données offre une grande souplesse de développement en permettant la mise à jour complète du contenu d'une application sans recompiler celle-ci.

Parmi les formats les plus couramment utilisés nous pouvons citer le texte ainsi que le XML mais aussi un format brut comme le binaire. Nous reviendrons en détail sur la manipulation de données binaire au cours du chapitre 20 intitulé *ByteArray*.

ActionScript 3 offre la possibilité de charger et d'envoyer ces différents types de données grâce à la classe `flash.net.URLLoader` qui remplace l'objet `LoadVars` ainsi que les différentes fonctions `loadVariables`, `loadVariablesNum` utilisées dans les précédentes versions d'ActionScript.

Nous retrouvons ici l'intérêt d'ActionScript 3 consistant à centraliser les fonctionnalités du lecteur.

Il est important de noter que contrairement à la classe `Loader`, la classe `URLLoader` diffuse directement les événements et ne dispose pas d'objet `LoaderInfo` interne. Ainsi l'écoute des différents événements se fait directement auprès de l'objet `URLLoader`.

En revanche, les événements diffusés sont quasiment les mêmes que la classe `LoaderInfo`. Voici le détail de chacun d'entre eux :

- `Event.OPEN` : diffusé lorsque le lecteur commence à charger les données.
- `ProgressEvent.PROGRESS` : diffusé lorsque le chargement est en cours. Celui-ci renseigne sur le nombre d'octets chargés et totaux.
- `Event.COMPLETE` : diffusé lorsque le chargement est terminé.
- `HTTPStatusEvent.HTTP_STATUS` : indique le code d'état de la requête HTTP.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque le chargement échoue.
- `SecurityErrorEvent.SECURITY_ERROR` : diffusé lorsque le lecteur tente de charger des données depuis un domaine non autorisé.

Tout en gérant les différentes erreurs pouvant survenir lors du chargement de données, nous allons commencer notre apprentissage en chargeant de simples données au format texte, puis XML.

Puis nous traiterons en détail l'envoi et réception de variables mais aussi de fichiers grâce aux classes `flash.net.FileReference` et `flash.net.FileReferenceList`.

---

## A retenir

---

- Les fonctions et méthodes `loadVariables`, `loadVariablesNum` et la classe `LoadVars` sont remplacées par la classe `URLLoader`.
- La classe `URLLoader` permet de charger des données au format texte, XML et binaire.

## Charger du texte

Afin de charger des données nous créons une instance de la classe `URLLoader`, puis nous utilisons la méthode `load` dont voici la signature :

```
public function load(request:URLRequest):void
```

Comme nous l'avons vu lors du précédent chapitre intitulé chargement de contenu, toute url doit être spécifiée au sein d'un objet `flash.net.URLRequest`. A l'aide d'un éditeur tel le bloc notes, nous créons un fichier texte nommé `donnees.txt` ayant le contenu suivant :

Voici le contenu du fichier texte !

Dans un nouveau document Flash CS3, nous créons une instance de la classe `URLLoader` puis nous chargeons le fichier texte :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// chargement des données
chargeurDonnees.load ( new URLRequest ("donnees.txt") );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementTermine ( pEvt:Event ):void
{
    trace("données chargées");
}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 0
    trace("code HTTP : " + pEvt.status);
}
```

```
function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}
```

A l'instar de la classe `flash.display.LoaderInfo` la classe `URLLoader` diffuse deux événements, une fois les données chargées. L'événement `HTTPStatusEvent.HTTP_STATUS` puis l'événement `Event.COMPLETE`.

---

Souvenez-vous qu'en local la propriété `status` de l'objet événementiel `HTTPStatusEvent` vaut toujours 0, même si le chargement échoue.

---

Dans le code suivant, nous chargeons le même fichier texte depuis un serveur, la propriété `status` de l'objet événementiel renvoie 200 :

```
// chargement des données
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// chargement des données
chargeurDonnees.load ( new URLRequest
("http://www.monserveur.org/donnees.txt" ) );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementTermine ( pEvt:Event ):void
{
    trace("données chargées");
}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 200
    trace("code HTTP : " + pEvt.status);
}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}
```



}

Si le lecteur ne parvient pas à charger le fichier distant, l'événement `IoErrorEvent.IO_ERROR` est diffusé ainsi que l'événement `HTTPStatusEvent.HTTP_STATUS`. Dans ce cas la propriété `status` contient le code d'erreur HTTP permettant de connaître la raison de l'échec.

---

Pour un tableau récapitulatif des différents codes d'erreurs possibles reportez-vous au chapitre 13 intitulé *Chargement de contenu*.

---

Afin d'accéder aux données chargées nous utilisons la propriété `data` de l'objet `URLLoader` :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// chargement des données
chargeurDonnees.load ( new URLRequest ("donnees.txt") );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IoErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IoErrorEvent.IO_ERROR, erreurChargement );

function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var contenu:String = pEvt.target.data;

    // affiche : Voici le contenu du fichier texte !
    trace( contenu );
}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 0
    trace("code HTTP : " + pEvt.status);
}

function erreurChargement ( pEvt:IoErrorEvent ):void
{
    trace("erreur de chargement");
}
```

```
| }
```

La propriété `dataFormat` de l'objet `URLLoader` permet de définir quel format de données nous chargeons. Celle-ci a la valeur `text` par défaut :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// affiche : text
trace( chargeurDonnees.dataFormat );
```

Il est recommandé pour des questions de lisibilité de code et de travail en équipe de toujours spécifier le type de données que nous chargeons même si il s'agit de données texte.

Pour cela nous utilisons trois propriétés constantes de la classe `flash.net.URLLoaderDataFormat`.

Voici le détail de chacune d'entre elles :

- `URLLoaderDataFormat.BINARY` : permet de charger des données au format binaire.
- `URLLoaderDataFormat.TEXT` : permet de charger du texte brut.
- `URLLoaderDataFormat.VARIABLES` : permet de charger des données url encodées.

Ainsi, même si nous souhaitons charger du texte, nous préférons l'indiquer pour des questions de lisibilité :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// nous souhaitons charger des données texte
chargeurDonnees.dataFormat = URLLoaderDataFormat.TEXT;

// chargement des données
chargeurDonnees.load ( new URLRequest ("donnees.txt") );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var contenu:String = pEvt.target.data;

    // affiche : Voici le contenu du fichier texte !
    trace( contenu );
```

```

        // affiche : text
        trace( pEvt.target.dataFormat );

    }

    function codeHTTP ( pEvt:HTTPStatusEvent ):void
    {

        // affiche : 0
        trace("code HTTP : " + pEvt.status);

    }

    function erreurChargement ( pEvt:IOErrorEvent ):void
    {

        trace("erreur de chargement");

    }

```

Dans les précédentes versions d'ActionScript la classe `LoadVars` était utilisée afin de charger des données externes. Celle-ci possédait un événement `onData` nous permettant de récupérer les données chargées brutes, sans passer par une interprétation des données chargées.

L'équivalent n'existe plus en ActionScript 3. Si nous souhaitons charger des données brutes, nous utilisons le format `URLLoaderDataFormat.BINARY`.

Dans certains cas, le chargement de fichier texte peut être utile, en particulier lors de chargement de fichiers comme le CSV.

---

Le CSV est un format simple de représentation de données sous forme de valeurs séparées par des virgules. Il est couramment utilisé en format d'export de logiciels comme Microsoft Excel ou Microsoft Outlook.

---

Dans l'exemple suivant nous chargeons un fichier CSV exporté depuis Microsoft Excel contenant des statistiques.

Voici un aperçu du contenu du fichier :

```

100
133.46
144.02
148
94.04
87.17
92.27
96.83
98.81

```

113.8  
113.2

Dans le code suivant nous chargeons le fichier CSV en tant que données texte, puis nous transformons la chaîne en un tableau à l'aide de la méthode `split` de la classe `String` :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// nous souhaitons charger des données texte
chargeurDonnees.dataFormat = URLLoaderDataFormat.TEXT;

// chargement des données
chargeurDonnees.load ( new URLRequest ("donnees.csv") );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var contenu:String = pEvt.target.data;

    // transformation de la chaîne en un tableau en séparant les données à
    // chaque saut de ligne
    var tableauDonnees:Array = contenu.split("\n");

    // affiche : 100
    trace( tableauDonnees[0] );

    // affiche : 94
    trace( tableauDonnees.length );
}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 0
    trace("code HTTP : " + pEvt.status);
}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}
```

Nous calculons l'amplitude du graphique en extrayant les valeurs minimum et maximum :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// nous souhaitons charger des données texte
chargeurDonnees.dataFormat = URLLoaderDataFormat.TEXT;

// chargement des données
chargeurDonnees.load ( new URLRequest ( "donnees.csv" ) );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

var hauteurMaximum:Number = 180;

function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var contenu:String = pEvt.target.data;

    // transformation de la chaîne en un tableau en séparant les données à
    // chaque saut de ligne
    var tableauDonnees:Array = contenu.split("\n");

    // extraction des valeurs minimum et maximum
    var valeurMinimum:Number = calculeMinimum ( tableauDonnees );
    var valeurMaximum:Number = calculeMaximum ( tableauDonnees );

    // calcul de l'amplitude du graphique
    var ratio:Number = hauteurMaximum / ( valeurMaximum - valeurMinimum );

    // affiche : 1.2699308593198815
    trace( ratio );
}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 0
    trace("code HTTP : " + pEvt.status);
}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}
```

```
function calculeMaximum ( pTableau:Array ):Number
{
    var lng:int = pTableau.length;
    var valeurMaximum:Number = Number.MIN_VALUE;

    while ( lng-- ) valeurMaximum = Math.max ( valeurMaximum, pTableau[lng] );

    return valeurMaximum;
}

function calculeMinimum ( pTableau:Array ):Number
{
    var lng:int = pTableau.length;
    var valeurMinimum:Number = Number.MAX_VALUE;

    while ( lng-- ) valeurMinimum = Math.min ( valeurMinimum, pTableau[lng] );

    return valeurMinimum;
}
```

Puis nous dessinons le graphique à l'aide de la fonction `dessineGaphique` :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// nous souhaitons charger des données texte
chargeurDonnees.dataFormat = URLLoaderDataFormat.TEXT;

// chargement des données
chargeurDonnees.load ( new URLRequest ( "donnees.csv" ) );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

var hauteurMaximum:Number = 180;

function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var contenu:String = pEvt.target.data;

    // affiche : Voici le contenu du fichier texte !
    var tableauDonnees:Array = contenu.split("\n");

    // extraction des valeurs minimum et maximum
    var valeurMinimum:Number = calculeMinimum ( tableauDonnees );
```

```

    var valeurMaximum:Number = calculeMaximum ( tableauDonnees );

    // calcul de l'amplitude du graphique
    var ratio:Number = hauteurMaximum / ( valeurMaximum - valeurMinimum );

    // dessine le graphique
    dessineGraphique ( ratio, valeurMaximum, tableauDonnees );

}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 0
    trace("code HTTP : " + pEvt.status);
}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}

function calculeMaximum ( pTableau:Array ):Number
{
    var lng:int = pTableau.length;
    var valeurMaximum:Number = Number.MIN_VALUE;

    while ( lng-- ) valeurMaximum = Math.max ( valeurMaximum, pTableau[lng] );

    return valeurMaximum;
}

function calculeMinimum ( pTableau:Array ):Number
{
    var lng:int = pTableau.length;
    var valeurMinimum:Number = Number.MAX_VALUE;

    while ( lng-- ) valeurMinimum = Math.min ( valeurMinimum, pTableau[lng] );

    return valeurMinimum;
}

var courbe:Shape = new Shape();

courbe.graphics.lineStyle ( 1, 0x990000, 1 );

var conteneurGraphique:Sprite = new Sprite();

conteneurGraphique.addChild( courbe );

conteneurGraphique.opaqueBackground = 0xFCEEBC;

```

```

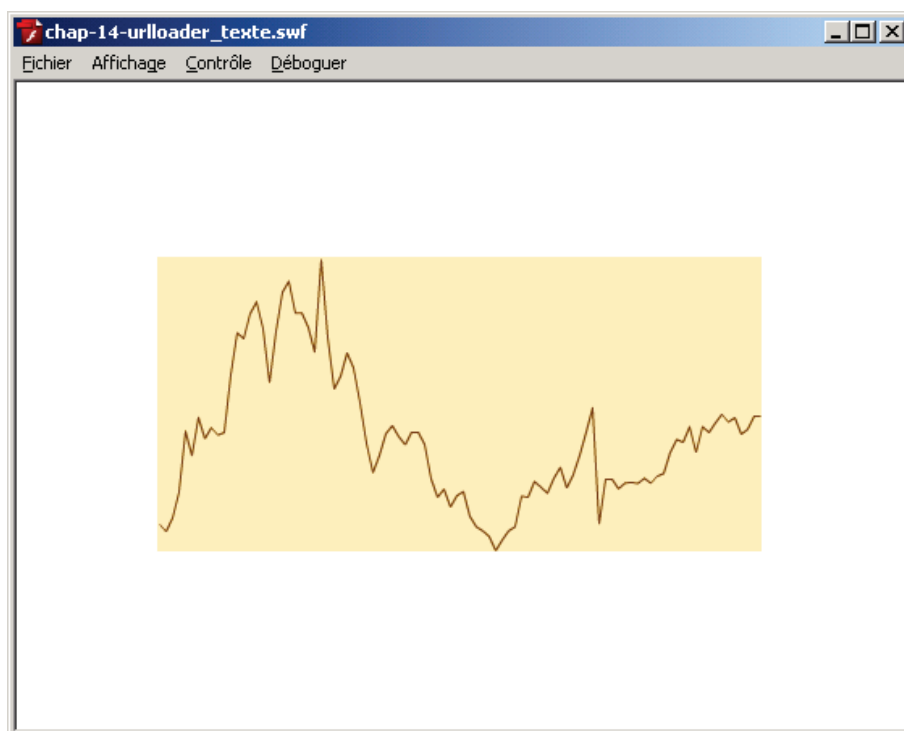
addChild ( conteneurGraphique );

function dessineGraphique ( pRatio:Number, pMaximum:Number, pDonnees:Array
):void
{
    for ( var i:int = 0; i< pDonnees.length; i++ )
    {
        if ( i == 0 ) courbe.graphics.moveTo ( i, (pMaximum-pDonnees[0]) *
pRatio );
        else courbe.graphics.lineTo ( i * 4, (pMaximum-pDonnees[i]) * pRatio
);
    }

    // centre le graphique
    conteneurGraphique.x = ( stage.stageWidth - conteneurGraphique.width ) / 2;
    conteneurGraphique.y = ( stage.stageHeight - conteneurGraphique.height ) /
2;
}

```

La figure 14-1 illustre le graphique généré depuis les données extraites du fichier CSV :



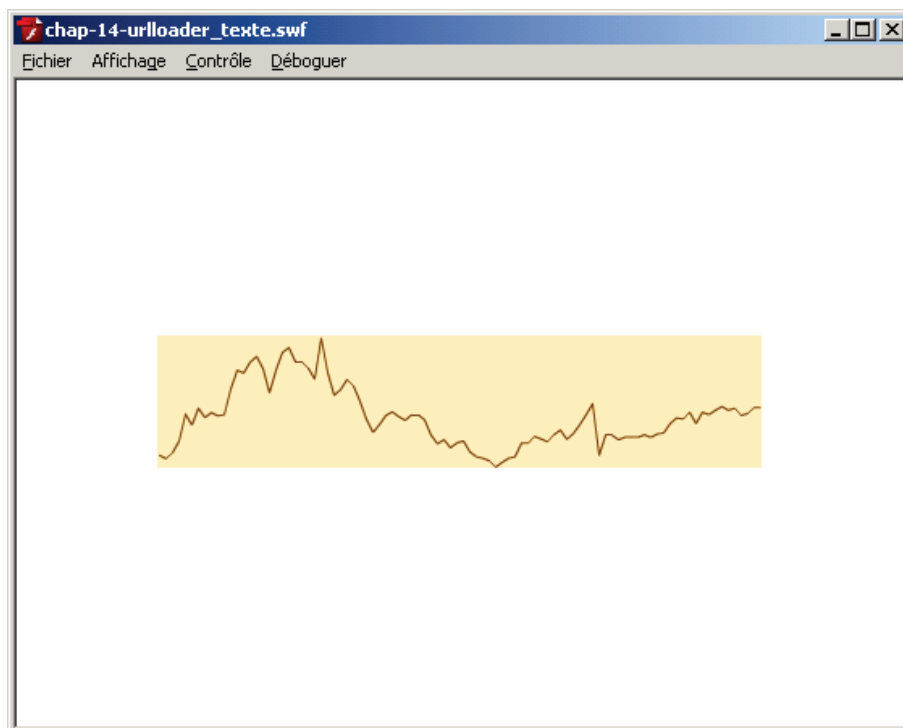
*Figure 14-1. Courbe de statistique.*

Si nous modifions la valeur de la variable `hauteurMaximum` :



```
var hauteurMaximum:Number = 80;
```

Nous voyons que le graphique est dessiné en conservant les proportions, comme l'illustre la figure 14-2 :



*Figure 14-2. Courbe de statistique réduite.*

Bien entendu, d'autres formats de données peuvent être utilisés. Nous allons nous intéresser au chargement de variables encodées au format URL.

## A retenir

- Afin de charger des données au format texte, nous passons la valeur `URLLoaderDataFormat.TEXT` à la propriété `dataFormat`.
- L'objet `URLLoader` ne possède pas d'objet `LoaderInfo` interne.
- Afin d'accéder aux données chargées, nous utilisons la propriété `data` de l'objet `URLLoader`.

## L'encodage URL

L'encodage URL est le standard de transmission d'informations par formulaire. Il permet de rendre compatible différents paramètres avec le protocole HTTP.

L'encodage respecte les règles suivantes :

- Les champs sont séparés par des esperluettes (caractère &)

- Un champ comporte un nom de champ, suivi de = puis de la valeur.
- Les espaces sont remplacés par des +.
- Les caractères non alphanumériques sont remplacés par %XX où XX représente le code ASCII en hexadécimal du caractère.

L'encodage URL va être utilisé au cours des prochaines parties afin de charger ou d'envoyer des variables.

## Charger des variables

Le chargement de données texte brut convient lorsque nous chargeons des données non structurées. Pour des données plus détaillées nous pouvons utiliser le format d'encodage URL détaillé précédemment.

La chaîne suivante est une chaîne encodée URL :

```
| titre=Voici un titre&contenu=Voici le contenu !
```

Nous allons placer la chaîne précédente au sein d'un fichier texte nommé `donnees_url.txt`.

Au sein d'un nouveau document Flash CS3, nous chargeons le fichier texte à l'aide de la méthode `load` :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// nous souhaitons charger des données texte
chargeurDonnees.dataFormat = URLLoaderDataFormat.TEXT;

// chargement des données
chargeurDonnees.load ( new URLRequest ( "donnees_url.txt" ) );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );

// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );

// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var contenu:String = pEvt.target.data;

    // affiche : titre=Voici un titre&contenu=Voici le contenu !
    trace( contenu );
}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{

```

```
// affiche : 0
trace("code HTTP : " + pEvt.status);

}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}
```

Les données sont ainsi chargées en tant que données texte brut. Il nous est difficile d'accéder pour le moment à la valeur de chaque variable `titre` ou `contenu`.

Afin d'interpréter et d'extraire des données de la chaîne encodée, deux possibilités s'offrent à nous :

La première consiste à décoder les données texte à l'aide de la classe `flash.net.URLVariables`. En passant la chaîne encodée URL au constructeur de celle-ci nous la décodons afin d'extraire la valeur de chaque variable :

```
function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var contenu:String = pEvt.target.data;

    // décodage de la chaîne encodée url sous forme d'objet
    var variables:URLVariables = new URLVariables ( contenu );

    // itération sur chaque variable décodée
    /*
    affiche :
    titre --> Voici un titre
    contenu --> Voici le contenu !
    */
    for ( var p in variables ) trace( p, "--> " + variables[p] );
}
```

Une fois l'objet `URLVariables` créé, nous accédons à chaque variable par la syntaxe pointée.

---

Nous retrouvons ici le même comportement que la classe `LoadVars` des précédentes versions d'ActionScript. Les variables chargées deviennent des propriétés de l'objet `URLVariables`.

---

Dans le code suivant, nous accédons manuellement aux deux variables `titre` et `contenu` devenues propriétés :

```
function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var donnees:String = pEvt.target.data;

    // décodage de la chaîne url encodée sous forme d'objet
    var variables:URLVariables = new URLVariables ( donnees );

    var titre:String = variables.titre;

    // affiche : Voici un titre
    trace( titre );

    var contenu:String = variables.contenu;

    // affiche : Voici le contenu !
    trace( contenu );
}
```

Au sein de notre document, nous plaçons deux champs texte dynamique auxquels nous associons deux noms d'occurrence. Puis nous remplissons dynamiquement ces derniers avec le contenu chargé correspondant :

```
function chargementTermine ( pEvt:Event ):void
{
    // accès aux données chargées
    var donnees:String = pEvt.target.data;

    // décodage de la chaîne url encodée sous forme d'objet
    var variables:URLVariables = new URLVariables ( donnees );

    var titre:String = variables.titre;
    var contenu:String = variables.contenu;

    // affectation des données chargées au champs texte
    champTitre.text = titre;
    champContenu.text = contenu;
}
```

La figure 14-3 illustre le résultat :

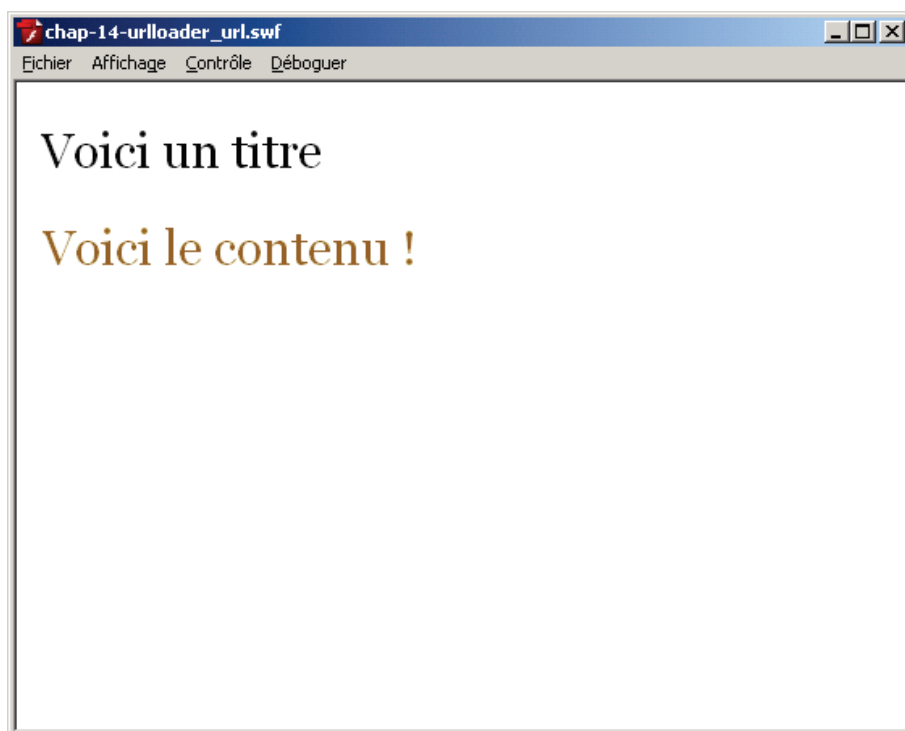


Figure 14-3. Contenu texte chargé dynamiquement.

Une seconde approche plus rapide consiste à spécifier au préalable à la propriété `dataFormat` la valeur `URLLoaderDataFormat.VARIABLES`.

Ainsi, les données chargées sont automatiquement décodées par un objet `URLVariables` :

```
// création de l'objet URLLoader
var chargeurDonnees:URLLoader = new URLLoader();

// nous souhaitons charger des données url encodées
chargeurDonnees.dataFormat = URLLoaderDataFormat.VARIABLES;

// chargement des données
chargeurDonnees.load ( new URLRequest ("donnees_url.txt") );

// écoute de l'événement Event.COMPLETE
chargeurDonnees.addEventListener( Event.COMPLETE, chargementTermine );
// écoute de l'événement HTTPStatusEvent.HTTP_STATUS
chargeurDonnees.addEventListener( HTTPStatusEvent.HTTP_STATUS, codeHTTP );
// écoute de l'événement IOErrorEvent.IO_ERROR
chargeurDonnees.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementTermine ( pEvt:Event ):void
{
    // accès à l'objet URLVariables
    var variables:URLVariables = pEvt.target.data;

    var titre:String = variables.titre;
```

```
var contenu:String = variables.contenu;

// affectation des données chargées au champs texte
champTitre.text = titre;
champContenu.text = contenu;
}

function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 0
    trace("code HTTP : " + pEvt.status);
}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}
```

Lorsque nous demandons explicitement de charger les données sous la forme de variables encodées URL, la propriété `data` de l'objet `URLLoader` retourne un objet `URLVariables` créé automatiquement par le lecteur Flash.

La classe `URLVariables` définit une méthode `decode` appelée en interne lors de l'interprétation de la chaîne encodée. Celle-ci peut aussi être appelée manuellement lorsque l'objet `URLVariables` est déjà créé et que nous souhaitons décoder une nouvelle chaîne.

Au cas où les variables chargées ne seraient pas formatées correctement, l'appel de celle-ci lève une erreur à l'exécution.

Dans le cas de la chaîne suivante :

```
| Voici un titre&contenu=Voici le contenu !
```

Nous avons omis volontairement la première variable `titre` associée au contenu `Voici un titre`. Si nous testons à nouveau le code précédent, l'erreur suivante est levée lors de l'événement `Event.COMPLETE` :

```
| Error: Error #2101: La chaîne transmise à URLVariables.decode() doit être une
| requête au format de code URL contenant des paires nom/valeur.
```

Si le texte chargé contient le caractère `&` et que nous ne souhaitons pas l'interpréter comme séparateur mais comme caractère composant une chaîne, nous pouvons indiquer son code caractère ASCII au format hexadécimal à l'aide du symbole `%`.

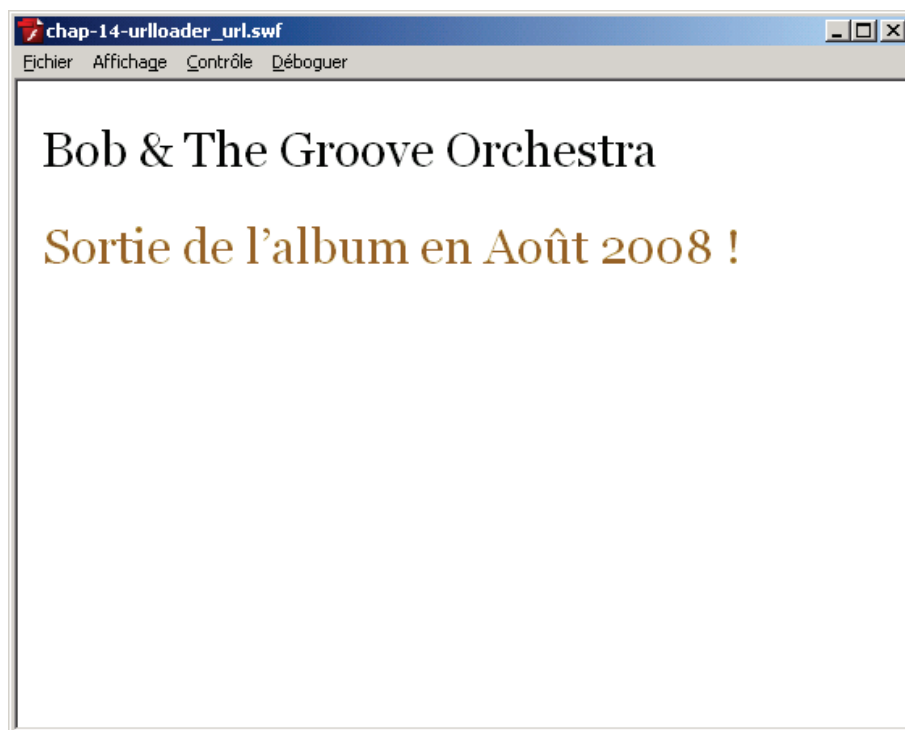
Ainsi, pour ajouter comme titre la chaîne de caractère suivante :

Bobby & The Groove Orchestra

Nous remplaçons le caractère & par son code caractère ASCII de la manière suivante :

```
titre=Bob %26 The Groove Orchestra&contenu=Sortie de l'album en Août 2008 !
```

La figure 14-4 illustre le résultat :



*Figure 14-4. Caractère spécial encodé en hexadécimal.*

Malheureusement, pour des questions de lisibilité et d'organisation, le format encodé URL s'avère rapidement limité. Nous préférons l'utilisation du format XML qui s'avère être un format standard et universel adapté à la représentation de données complexes.

**A retenir**

- Afin de décoder une chaîne encodée URL nous utilisons un objet `URLVariables`.
- Afin de charger des variables encodées URL, nous passons la valeur `URLLoaderDataFormat.VARIABLES` à la propriété `dataFormat`.
- Afin de décoder une chaîne de caractères encodée URL nous pouvons la passer au constructeur de la classe `URLVariables` où à la méthode `decode`.

## Charger des données XML

Si nous devons représenter une petite partie de la discographie de Stevie Wonder au format encodé URL nous obtiendrons la chaîne suivante :

```
&album_1=Talking Book&artiste_1=Stevie Wonder&label_1=Motown&annee_1=1972&album_2=Songs in the key of life&artiste_2=Stevie Wonder&label_2=Motown&annee_2=1976
```

Les mêmes données au format XML seraient formatées de la manière suivante :

```
<DISCOGRAPHIE>
  <ALBUM>
    <TITRE>
      Talking Book
    </TITRE>
    <ANNEE DATE="1972"/>
    <ARTISTE NOM="Wonder" PRENOM="Stevie"/>
    <LABEL NOM="Motown"/>
  </ALBUM >
  <ALBUM>
    <TITRE>
      Songs in the key of life
    </TITRE>
    <ANNEE DATE="1976"/>
    <ARTISTE NOM="Wonder" PRENOM="Stevie"/>
    <LABEL NOM="Motown"/>
  </ALBUM >
</DISCOGRAPHIE>
```

La représentation XML est plus naturelle et plus adaptée dans le cas de données structurées.

Nous avons manipulé le format XML au cours du chapitre 2 intitulé *Langage et API du lecteur Flash*. Nous allons découvrir comment charger dynamiquement un fichier XML afin de construire une interface graphique.

En ActionScript 3, la classe XML ne fait plus partie de l'API du lecteur Flash, mais appartient au langage ActionScript 3 reposant sur ECMAScript, c'est donc une classe *haut niveau*.



---

La classe XML ne gère donc plus le chargement de données comme c'était le cas dans les précédentes versions d'ActionScript. Afin de charger des données XML, nous chargeons simplement une chaîne de caractères sous la forme de texte brut, puis nous la transformons en objet XML.

---

Dans un nouveau document Flash CS3, nous associons la classe de document suivante :

```
package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault

    {

        public function Document ()

        {

        }

    }

}
```

A côté du document Flash en cours nous sauvons un fichier XML sous le nom `donnees.xml` contenant les données suivantes :

```
<MENU>
<BOUTON legende="Accueil" couleur="0x887400" vitesse="1" swf="accueil.swf"/>
<BOUTON legende="Photos" couleur="0x005587" vitesse="1" url="photos.swf"/>
<BOUTON legende="Blog" couleur="0x125874" vitesse="1" url="blog.swf"/>
<BOUTON legende="Liens" couleur="0x59CCAA" vitesse="1" url="liens.swf"/>
<BOUTON legende="Forum" couleur="0xEE44AA" vitesse="1" url="forum.swf"/>
</MENU>
```

Puis nous le chargeons :

```
package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.events.HTTPStatusEvent;
    import flash.events.IOErrorEvent;

    public class Document extends ApplicationDefault

    {
```

```

private var chargeur:URLLoader;

public function Document ()
{
    chargeur = new URLLoader();

    chargeur.dataFormat = URLLoaderDataFormat.TEXT;

    chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
    chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS, codeHTTP
);
    chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );

    chargeur.load ( new URLRequest ("donnees.xml") );
}

private function chargementTermine ( pEvt:Event ) :void
{
    var donneesXML:XML = new XML ( pEvt.target.data );

    /*
    affiche :
    <MENU>
        <BOUTON legende="Accueil" couleur="0x887400" vitesse="1"
url="accueil.swf"/>
        <BOUTON legende="Photos" couleur="0x005587" vitesse="1"
url="photos.swf"/>
        <BOUTON legende="Blog" couleur="0x125874" vitesse="1"
url="blog.swf"/>
        <BOUTON legende="Liens" couleur="0x59CCAA" vitesse="1"
url="liens.swf"/>
        <BOUTON legende="Forum" couleur="0xEE44AA" vitesse="1"
url="forum.swf"/>
    </MENU>
    */
    trace( donneesXML );
}

private function codeHTTP ( pEvt:HTTPStatusEvent ):void
{
    // affiche : 0
    trace("code HTTP : " + pEvt.status);
}

private function erreurChargement ( pEvent:IOErrorEvent ):void
{
    trace("erreur de chargement");
}

```

```
}  
}
```

Lorsque la méthode écouteur `chargementTermine` se déclenche nous accédons aux données chargées puis nous transformons la chaîne de caractères en objet XML.

Nous allons reprendre le menu construit lors du chapitre 10 intitulé *Diffusion d'événements personnalisés* afin de charger dynamiquement les données du menu depuis un fichier XML.

Afin de créer notre menu, nous reprenons la classe `Bouton` utilisée lors du chapitre 10 puis nous l'importons ainsi que la classe `Sprite` :

```
package org.bytearray.document  
  
{  
  
    import org.bytearray.abstrait.ApplicationDefault;  
    import org.bytearray.ui.Bouton;  
    import flash.net.URLLoader;  
    import flash.net.URLLoaderDataFormat;  
    import flash.net.URLRequest;  
    import flash.events.Event;  
    import flash.events.HTTPStatusEvent;  
    import flash.events.IOErrorEvent;  
    import flash.display.Sprite;  
  
    public class Document extends ApplicationDefault  
    {  
  
        private var chargeur:URLLoader;  
        private var conteneurMenu:Sprite;  
  
        public function Document ()  
        {  
  
            conteneurMenu = new Sprite();  
  
            addChild ( conteneurMenu );  
  
            chargeur = new URLLoader();  
  
            chargeur.dataFormat = URLLoaderDataFormat.TEXT;  
  
            chargeur.addEventListener ( Event.COMPLETE, chargementTermine );  
            chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS, codeHTTP  
);  
            chargeur.addEventListener ( IOErrorEvent.IO_ERROR,  
erreurChargement );  
  
            chargeur.load ( new URLRequest ( "donnees.xml" ) );  
  
        }  
  
        private function chargementTermine ( pEvt:Event ) :void
```

```

    {
        var donneesXML:XML = new XML ( pEvt.target.data );

        creerMenu ( donneesXML );
    }

    private function codeHTTP ( pEvt:HTTPStatusEvent ):void
    {
        // affiche : 0
        trace("code HTTP : " + pEvt.status);
    }

    private function erreurChargement ( pEvent:IOErrorEvent ):void
    {
        trace("erreur de chargement");
    }

    private function creerMenu ( pXML:XML ):void
    {
        var i:int = 0;
        var monBouton:Bouton;

        for each ( var enfant:XML in pXML.* )
        {
            // récupération des infos
            var legende:String = enfant.@legende;
            var couleur:Number = enfant.@couleur;
            var vitesse:Number = enfant.@vitesse;
            var swf:String = enfant.@url;

            // création des boutons
            monBouton = new Bouton( 60, 40, swf, couleur, vitesse,
legende );

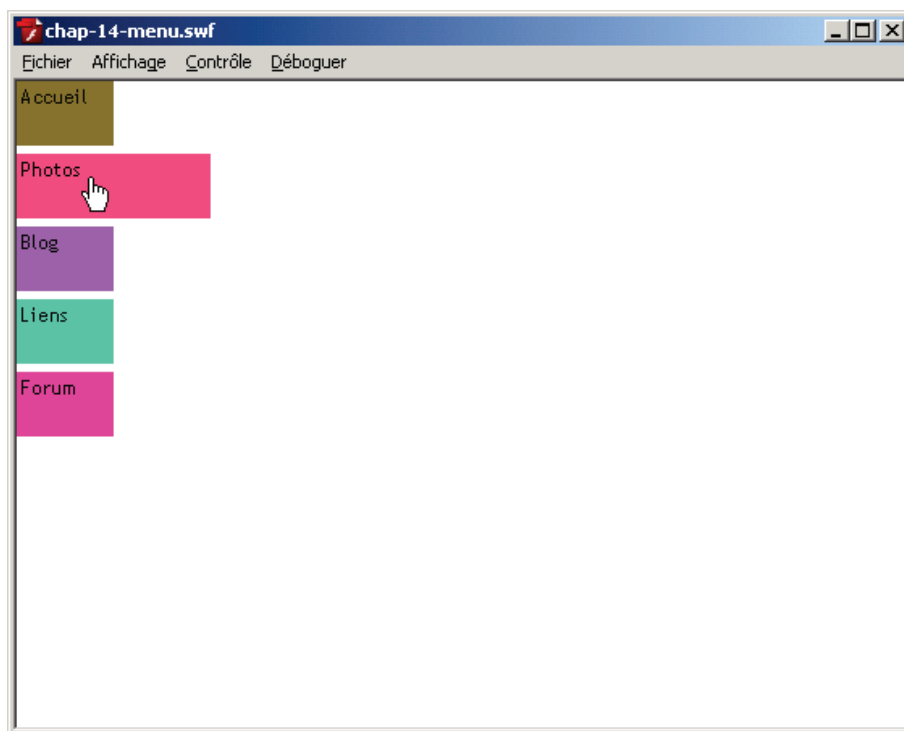
            // positionnement
            monBouton.y = (monBouton.height + 5) * i;

            // ajout à la liste d'affichage
            conteneurMenu.addChild ( monBouton );

            i++;
        }
    }
}

```

La figure 14-4 illustre le résultat :



*Figure 14-4. Menu dynamique XML.*

Afin d'écouter chaque clic bouton, nous importons la classe `EvenementBouton` créée lors du chapitre 10 puis nous ciblons l'événement `EvenementBouton.CLICK` en utilisant la phase de capture :

```
package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefaut;
    import org.bytearray.ui.Bouton;
    import org.bytearray.evenements.EvenementBouton;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.events.HTTPStatusEvent;
    import flash.events.IOErrorEvent;
    import flash.display.Sprite;

    public class Document extends ApplicationDefaut

    {

        private var chargeur:URLLoader;
        private var conteneurMenu:Sprite;

        public function Document ()
```

```

        {

            conteneurMenu = new Sprite();

            addChild ( conteneurMenu );

            conteneurMenu.addEventListener ( EvenementBouton.CLICK,
clicBouton, true );

            chargeur = new URLLoader();

            chargeur.dataFormat = URLLoaderDataFormat.TEXT;

            chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
            chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS, codeHTTP
);
            chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );

            chargeur.load ( new URLRequest ( "donnees.xml" ) );

        }

        private function chargementTermine ( pEvt:Event ) :void
        {

            var donneesXML:XML = new XML ( pEvt.target.data );

            creerMenu ( donneesXML );

        }

        private function codeHTTP ( pEvt:HTTPStatusEvent ) :void
        {

            // affiche : 0
            trace("code HTTP : " + pEvt.status);

        }

        private function erreurChargement ( pEvent:IOErrorEvent ) :void
        {

            trace("erreur de chargement");

        }

        private function creerMenu ( pXML:XML ) :void
        {

            var i:int = 0;
            var monBouton:Bouton;

            for each ( var enfant:XML in pXML.* )

            {

                // récupération des infos

```

```

        var legende:String = enfant.@legende;
        var couleur:Number = enfant.@couleur;
        var vitesse:Number = enfant.@vitesse;
        var swf:String = enfant.@url;

        // création des boutons
        monBouton = new Bouton( 60, 40, swf, couleur, vitesse,
legende );

        // positionnement
        monBouton.y = (monBouton.height + 5) * i;

        // ajout à la liste d'affichage
        conteneurMenu.addChild ( monBouton );

        i++;
    }
}

private function clicBouton( pEvt:EvenementBouton ):void
{
    // affiche : photos.swf
    trace( pEvt.lien );
}
}
}

```

Nous avons ici réutilisé la classe `Bouton` créée lors du chapitre 10, en prévoyant un modèle simple d'utilisation nous avons pu réutiliser cette classe sans aucun problème.

Afin de désactiver totalement notre menu, nous devons supprimer l'objet `conteneurMenu` de la liste d'affichage, puis passer sa référence à `null` :

```

package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefaut;
    import org.bytearray.ui.Button;
    import org.bytearray.events.ButtonEvent;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    import flash.events.Event;
    import flash.events.HTTPStatusEvent;
    import flash.events.IOErrorEvent;
    import flash.events.MouseEvent;
    import flash.display.Sprite;

    public class Document extends ApplicationDefaut

```

```

{

    private var chargeur:URLLoader;
    private var conteneurMenu:Sprite;

    public function Document ()

    {

        conteneurMenu = new Sprite();

        addChild ( conteneurMenu );

        stage.doubleClickEnabled = true;

        stage.addEventListener ( MouseEvent.DOUBLE_CLICK, desactive );

        conteneurMenu.addEventListener ( ButtonEvent.CLICK, clicBouton,
true );

        chargeur = new URLLoader();

        chargeur.dataFormat = URLLoaderDataFormat.TEXT;

        chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
        chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS, codeHTTP
);
        chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );

        chargeur.load ( new URLRequest ("donnees.xml") );

    }

    private function chargementTermine ( pEvt:Event ) :void

    {

        var donneesXML:XML = new XML ( pEvt.target.data );

        creerMenu ( donneesXML );

    }

    private function codeHTTP ( pEvt:HTTPStatusEvent ) :void

    {

        // affiche : 0
        trace("code HTTP : " + pEvt.status);

    }

    private function erreurChargement ( pEvent:IOErrorEvent ) :void

    {

        trace("erreur de chargement");

    }

    private function creerMenu ( pXML:XML ) :void

```



```

    {
        var i:int = 0;
        var monBouton:Button;

        for each ( var enfant:XML in pXML.* )
        {
            // récupération des infos
            var legende:String = enfant.@legende;
            var couleur:Number = enfant.@couleur;
            var vitesse:Number = enfant.@vitesse;
            var swf:String = enfant.@url;

            // création des boutons
            monBouton = new Button( 60, 40, swf, couleur, vitesse,
legende );

            // positionnement
            monBouton.y = (monBouton.height + 5) * i;

            // ajout à la liste d'affichage
            conteneurMenu.addChild ( monBouton );

            i++;
        }
    }

    private function clicBouton( pEvt:MouseEvent ):void
    {
        // affiche : photos.swf
        trace( pEvt.lien );
    }

    private function desactive ( pEvt:MouseEvent ):void
    {
        removeChild ( conteneurMenu );
        conteneurMenu = null;
    }
}

```

Souvenez-vous que pour correctement désactiver un élément interactif, nous devons supprimer toutes ses références.

Dans cet exemple, les boutons sont seulement référencés de par leur présence au sein de l'objet graphique `conteneurMenu`. En désactivant ce dernier nous rendons alors inaccessible ses enfants. Ces

derniers deviennent ainsi éligible à la suppression par le ramasse miettes.

Nous avons vu lors du chapitre précédent que le modèle de sécurité du lecteur intégrait des restrictions concernant le chargement de contenu externe. Nous allons au cours de la partie suivante nous intéresser aux restrictions de sécurité dans un contexte de chargement de données.

### A retenir

- Afin de charger un flux XML, nous passons la valeur `URLLoaderDataFormat.TEXT` à la propriété `dataFormat` de l'objet `URLLoader`.
- La classe XML ne s'occupe plus du chargement du flux XML.
- Une fois la chaîne de caractères chargée par l'objet `URLLoader`, nous créons un objet XML à partir de celle-ci.

## Chargement de données et sécurité

Le chargement de données est soumis aux mêmes restrictions de sécurité que le chargement de contenu.

Imaginons le scénario suivant :

Vous devez développer une application Flash permettant de lire des flux XML provenant de différents blogs. Cette application sera hébergée sur votre serveur et ne peut pour des raisons de sécurité accéder aux flux distants.

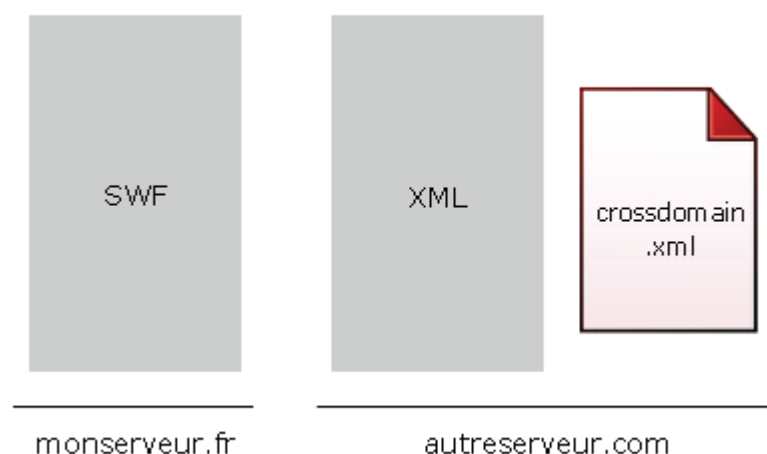
Au cours du chapitre 13, nous avons vu qu'il était possible d'autoriser de trois manières un SWF tentant de charger du contenu depuis un serveur distant :

- Par un fichier de régulation (XML).
- Par l'appel de la méthode `allowDomain` de la classe `flash.system.Security` dans le SWF à charger.
- Par l'utilisation d'un fichier de proxy.

Attention, dans notre cas, nous ne chargeons plus de contenu SWF. Il est donc impossible d'appeler la méthode `allowDomain` de l'objet `Security`.

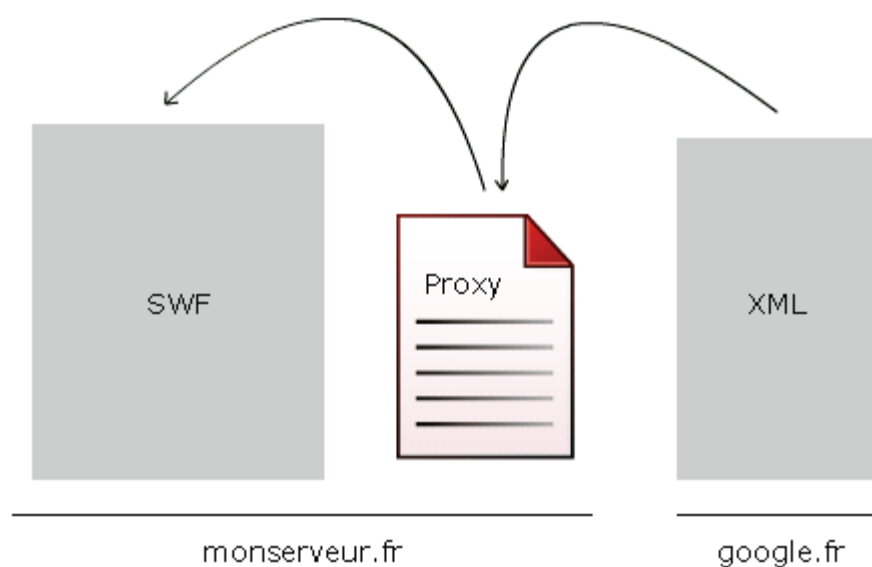
Ainsi, dans le cas de chargement de flux XML, texte, ou autres seules deux méthodes d'autorisations s'offrent à vous :

Par un fichier de régulation XML comme l'illustre la figure 14-5 :



*Figure 14-5. Autorisation par fichier de régulation.*

Ou bien par l'utilisation d'un fichier proxy :



*Figure 14-6. Utilisation d'un proxy.*

Reportez vous au chapitre 13 intitulé chargement de contenu pour plus d'informations relatives au modèle de sécurité du lecteur Flash 9 et l'utilisation de fichier de régulation ou proxy.

## A retenir

- Les mêmes restrictions de sécurité liées au chargement de contenu, s'appliquent lors du chargement de données.
- Il est impossible de placer au sein des données chargées, un appel à la méthode `allowDomain` de l'objet `Security`.

## Charger des données binaires

La grande puissance du lecteur Flash 9 en ActionScript 3 réside dans la lecture de données au format binaire grâce à la classe bas niveau `flash.utils.ByteArray`. Afin de charger des données binaires brutes, nous devons passer à la propriété `dataFormat` la valeur `URLLoaderDataFormat.BINARY`.

Dans un nouveau document Flash CS3, nous chargeons un fichier PSD en associant une classe du document contenant le code suivant :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.URLRequest;
    import flash.utils.ByteArray;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.events.Event;
    import flash.events.HTTPStatusEvent;
    import flash.events.IOErrorEvent;

    public class Document extends ApplicationDefault
    {
        private var chargeur:URLLoader;

        public function Document ()
        {
            chargeur = new URLLoader();

            chargeur.dataFormat = URLLoaderDataFormat.BINARY;

            chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
            chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS, codeHTTP
        );
            chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
            erreurChargement );

            chargeur.load ( new URLRequest ( "maquette.psd" ) );
        }

        private function chargementTermine ( pEvt:Event ) :void
        {

```

```

        var donneesBinaire:ByteArray = pEvt.target.data;

        // affiche : 182509
        trace( donneesBinaire.length );

    }

    private function codeHTTP ( pEvt:HTTPStatusEvent ):void
    {

        // affiche : 0
        trace("code HTTP : " + pEvt.status);

    }

    private function erreurChargement ( pEvent:IOErrorEvent ):void
    {

        trace("erreur de chargement");

    }

}

```

La variable `donneesBinaire` contient le flux binaire du fichier PSD chargé. En créant une classe `EntetePSD`, nous allons lire l'entête du fichier afin d'extraire différentes informations comme la taille du document, la version, ainsi que le modèle de couleur utilisé.

Pour cela, nous créons une classe `EntetePSD` contenant le code suivant :

```

package org.bytearray.psd

{

    import flash.utils.ByteArray;

    public class EntetePSD

    {

        private var flux:ByteArray;
        private var _signature:String;
        private var _version:int;
        private var _canal:int;
        private var _hauteur:int;
        private var _largeur:int;
        private var _profondeur:int;
        private var _mode:int;

        private static const MODES_COULEURS:Array = new Array ("Bitmap", "Mode
niveaux de gris", "Indexé", "RVB", "CMJN", "Multi Canal", "Deux tons",
"Lab");

        public function EntetePSD ( pFlux:ByteArray )

```

```

    {
        flux = pFlux;

        // extrait la signature de l'entête PSD (doit être 8BPS)
        _signature = flux.readUTFBytes(4);

        // extrait la version (doit être égal à 1)
        _version = flux.readUnsignedShort();

        // nous sautons 6 octets
        flux.position += 6;

        // extrait le canal utilisé
        _canal = flux.readUnsignedShort();

        // extrait la largeur du document
        _largeur = flux.readInt();

        // extrait la hauteur du document
        _hauteur = flux.readInt();

        // bpp
        _profondeur = flux.readUnsignedShort();

        // mode colorimétrique (Bitmap=0, Mode niveaux de gris=1,
        Indexé=2, RVB=3, CMJN=4, Multi Canal=7, Deux tons=8, Lab=9)
        _mode = flux.readUnsignedShort();
    }

    public function toString ( ):String
    {
        return "[EntetePSD signature : " + signature + ", version : " +
version + ", canal : " + canal + ", largeur : " +
        largeur + ", hauteur : " + hauteur + ", profondeur : " +
profondeur + ", mode colorimétrique : " + MODES_COULEURS [ mode ] +"]";
    }

    public function get signature ():String
    {
        return _signature;
    }

    public function get version ():int
    {
        return _version;
    }

    public function get canal ():int
    {

```

```
        return _canal;
    }

    public function get largeur ():int
    {
        return _largeur;
    }

    public function get hauteur ():int
    {
        return _hauteur;
    }

    public function get profondeur ():int
    {
        return _profondeur;
    }

    public function get mode ():int
    {
        return _mode;
    }
}
}
```

Le flux binaire généré par le lecteur Flash est passé au constructeur, puis nous lisons le flux à l'aide des méthodes de la classe `ByteArray`. Nous reviendrons sur celles-ci au cours du chapitre 20 intitulé *ByteArray*.

Afin d'extraire les informations du PSD nous instancions la classe `EntetePSD` en passant le flux binaire au constructeur :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefaut;
    import org.bytearray.psd.EntetePSD;
    import flash.net.URLRequest;
    import flash.utils.ByteArray;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.events.Event;
```

```

import flash.events.HTTPStatusEvent;
import flash.events.IOErrorEvent;

public class Document extends ApplicationDefault
{
    private var chargeur:URLLoader;

    public function Document ()
    {
        chargeur = new URLLoader();

        chargeur.dataFormat = URLLoaderDataFormat.BINARY;

        chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
        chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS, codeHTTP
    );
        chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );

        chargeur.load ( new URLRequest ( "maquette.psd" ) );
    }

    private function chargementTermine ( pEvt:Event ) :void
    {
        var donneesBinaire:ByteArray = pEvt.target.data;

        var infosPSD:EntetePSD = new EntetePSD( donneesBinaire );

        // affiche : [EntetePSD signature : 8BPS, version : 1, canal :
3, largeur : 450, hauteur : 562, profondeur : 8, mode colorimétrique : RVB]
        trace( infosPSD );
    }

    private function codeHTTP ( pEvt:HTTPStatusEvent ):void
    {
        // affiche : 0
        trace("code HTTP : " + pEvt.status);
    }

    private function erreurChargement ( pEvt:IOErrorEvent ):void
    {
        trace("erreur de chargement");
    }
}

```



Grâce à la classe `ByteArray` nous pouvons charger n'importe quel type de fichiers puis en extraire des informations. Nous pourrions imaginer une application RIA permettant d'héberger tout type de fichier. Celle-ci pourrait extraire les informations provenant de fichiers PSD, AI, FLA ou autres.

Nous pourrions optimiser la classe `EntetePSD` en diffusant un événement personnalisé `EvenementEntetePSD.INFO`. L'objet événementiel diffusé contiendrait toutes les propriétés nécessaires à la description du fichier PSD.

La classe `ByteArray` ouvre des portes à toutes sortes de possibilités. Nous reviendrons en détail sur la puissance de cette classe au cours du chapitre 20 intitulé *ByteArray*.

### A retenir

- Afin de charger un flux binaire, nous passons la valeur `URLLoaderDataFormat.BINARY` à la propriété `dataFormat` de l'objet `URLLoader`.
- Le flux binaire généré est représenté par la classe `flash.utils.ByteArray`.

## Concept d'envoi de données

Comme nous l'avons vu lors du chapitre 13 intitulé *Chargement de contenu*, toute URL doit être spécifiée au sein d'un objet `URLRequest`.

Celui-ci offre pourtant bien d'autres fonctionnalités que nous n'avons pas encore exploitées. Nous allons nous intéresser au cours des prochaines parties au concept d'envoi de données.

Pour cela, voyons en détail les propriétés de la classe `URLRequest` :

- `contentType` : type de contenu MIME des données envoyées en POST.
- `data` : contient les données à envoyer. Peut être de type `String`, `URLVariables` ou `ByteArray`.
- `method` : permet d'indiquer si les données doivent être envoyées par la méthode GET ou POST.
- `requestHeaders` : tableau contenant les entêtes HTTP définies par des objets `flash.net.URLRequestHeader`.
- `url` : contient l'url à atteindre.

Nous allons au cours des exercices suivants utiliser ces différentes propriétés afin de découvrir leurs intérêts.

Au sein du lecteur Flash nous pouvons distinguer trois types d'envoi de données :

- Envoi simple : les variables sont envoyées à l'aide d'une nouvelle fenêtre navigateur.
- Envoi discret : l'envoi des données est transparent pour l'utilisateur. Aucune fenêtre navigateur n'est ouverte durant l'envoi.
- Envoi discret avec validation : l'envoi des données est transparent, le lecteur Flash reçoit un retour du serveur permettant une validation d'envoi des données au sein de l'application Flash.

Nous allons traiter chacun des cas et comprendre les différences entre chaque approche.

## Envoyer des variables

Nous allons commencer par un envoi simple de données en développant un formulaire permettant d'envoyer un email depuis Flash. Ce type de développement peut être intégré dans une rubrique « Contactez-nous » au sein d'un site.

Nous verrons qu'il est possible sous certaines conditions d'envoyer un email depuis le lecteur Flash sans passer par un script serveur grâce à la classe `flash.net.Socket` que nous traiterons au cours du chapitre 19 intitulé *Les sockets*.

Par défaut, le lecteur Flash n'a pas la capacité d'envoyer un email de manière autonome. Afin d'y parvenir, nous devons passer les informations nécessaires à un script serveur afin que celui-ci puisse envoyer le message.

Dans un nouveau document Flash CS3, nous associons la classe du document suivante :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    public class Document extends ApplicationDefault
    {
        public function Document ()
        {
        }
    }
}
```

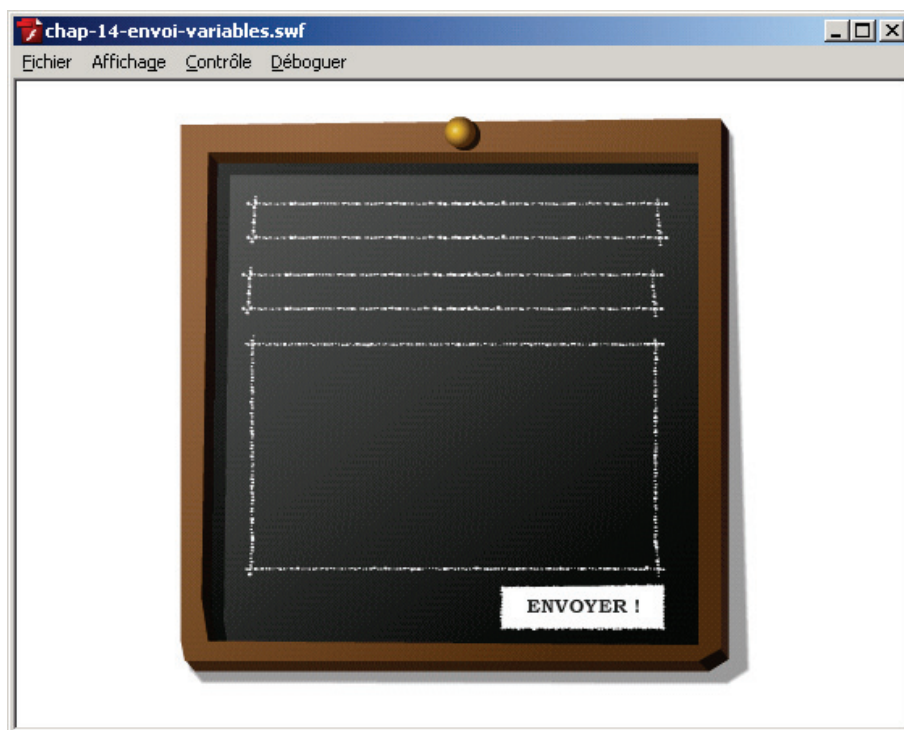
```

    }
}

```

Puis nous plaçons trois champs texte de saisie et un bouton afin de créer une interface d'envoi de mail.

La figure 14-7 illustre l'interface :



*Figure 14-7. Formulaire d'envoi d'email.*

Chaque instance est définie au sein de la classe du document :

```

package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.display.SimpleButton;
    import flash.text.TextField;

    public class Document extends ApplicationDefault
    {
        public var destinataire:TextField;
        public var sujet:TextField;
        public var message:TextField;
        public var boutonEnvoi:SimpleButton;

        public function Document ()
    }
}

```

```

        {

        }

    }

}

```

Nous allons créer à présent le code PHP nous permettant de réceptionner les variables transmises depuis Flash. Nous allons utiliser dans un premier temps la méthode GET.

Les variables sont ainsi accessible par l'intermédiaire du tableau associatif `$_GET` :

```

<?php

$destinataire = $_GET ["destinataire"];
$sujet = $_GET ["sujet"];
$message = $_GET ["message"];

if ( isset ( $destinataire ) && isset ( $sujet ) && isset ( $message ) )
{

    echo $destinataire. "<br>". $sujet. "<br>" . $message;

} else echo "Variables non transmises";

?>

```

Il convient de placer ce script sur un serveur local ou distant afin de pouvoir tester l'application. Dans notre cas, le script serveur est placé sur un serveur local. Le script est donc accessible en `localhost` :

```

| http://localhost/mail/envoiMail.php

```

Nous ajoutons à présent le code nécessaire afin d'envoyer les variables à notre script distant :

```

package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefaut;
    import flash.display.SimpleButton;
    import flash.text.TextField;
    import flash.net.navigateToURL;
    import flash.net.URLRequest;
    import flash.events.MouseEvent;

    public class Document extends ApplicationDefaut

    {

        public var destinataire:TextField;
        public var sujet:TextField;
        public var message:TextField;
        public var boutonEnvoi:SimpleButton;
    }
}

```

```

public function Document ()
{
    boutonEnvoi.addEventListener ( MouseEvent.CLICK, envoiMail );
}

private function envoiMail ( pEvt:MouseEvent ):void
{
    // affectation des variables à envoyer coté serveur
    var destinataireEmail:String = destinataire.text;
    var sujetEmail:String = sujet.text;
    var messageEmail:String = message.text;

    // création de l'objet URLRequest
    var requete:URLRequest = new URLRequest
("http://localhost/mail/envoiMail.php");

    // ouvre une nouvelle fenêtre navigateur et envoi les variables
    navigateToURL ( requete );
}
}
}

```

Nous stockons le destinataire, le sujet ainsi que le message au sein de trois variables. Puis nous les ajoutons en fin d'url du script distant de la manière suivante :

```

package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.display.SimpleButton;
    import flash.text.TextField;
    import flash.net.navigateToURL;
    import flash.net.URLRequest;
    import flash.events.MouseEvent;

    public class Document extends ApplicationDefault
    {
        public var destinataire:TextField;
        public var sujet:TextField;
        public var message:TextField;
        public var boutonEnvoi:SimpleButton;

        public function Document ()
        {
            boutonEnvoi.addEventListener ( MouseEvent.CLICK, envoiMail );
        }
    }
}

```

```
private function envoiMail ( pEvt:MouseEvent ):void
{
    // affectation des variables à envoyer coté serveur
    var destinataireEmail:String = destinataire.text;
    var sujetEmail:String = sujet.text;
    var messageEmail:String = message.text;

    var requete:URLRequest = new URLRequest
    ("http://localhost/mail/envoiMail.php?destinataire="+destinataireEmail+"&sujet=
    "+sujetEmail+"&message="+messageEmail);

    // ouvre une nouvelle fenêtre navigateur et envoi les variables
    navigateToURL ( requete );
}
}
```

Le point d'interrogation au sein de l'URL indique au navigateur que le texte suivant contient les variables représentées par des paires noms/valeurs séparées par des esperluettes (caractère &).

Une fois les informations saisies comme l'illustre la figure 14-8 :

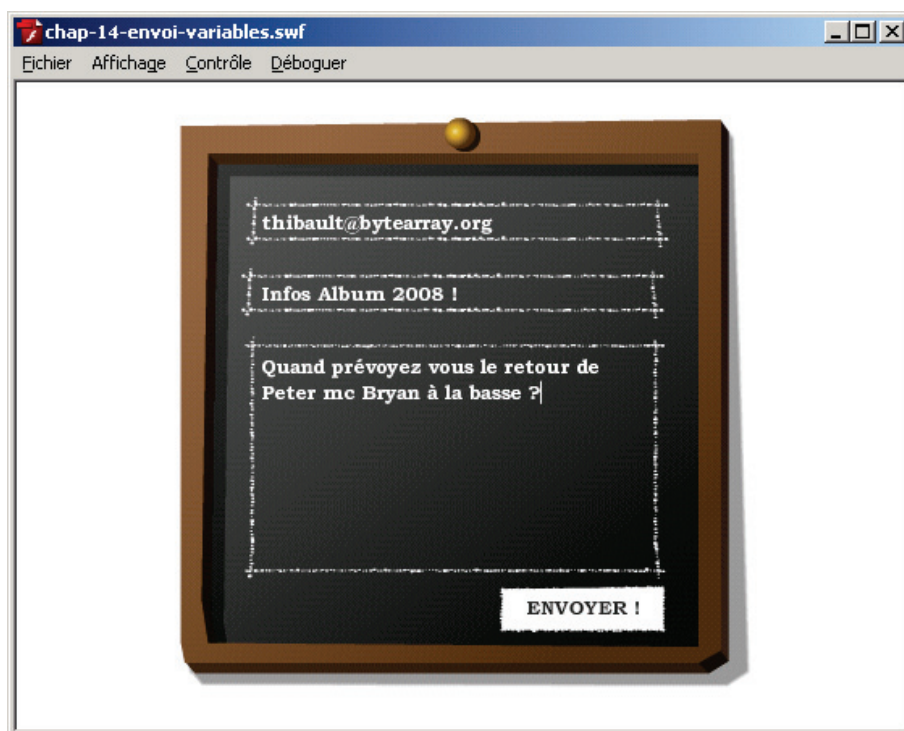
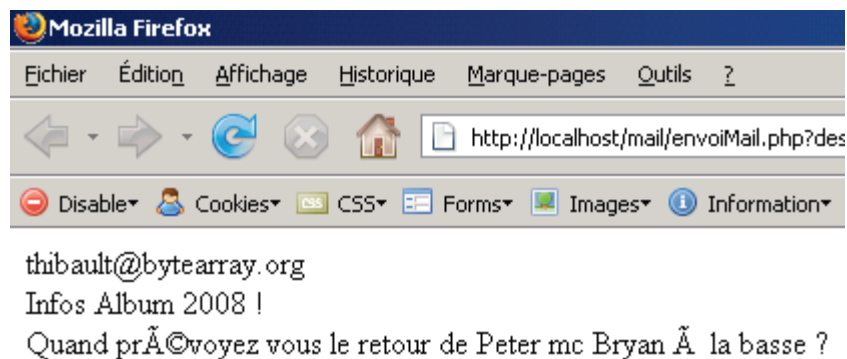


Figure 14-8. Formulaire d'envoi d'email.

Nous cliquons sur le bouton `boutonEnvoi`, une nouvelle fenêtre navigateur s'ouvre, les variables sont automatiquement placées en fin d'url :

```
http://localhost/mail/envoiMail.php?destinataire=thibault@bytearray.org& sujet
=Infos%20Album%202008%20!&message=Quand%20pr%C3%A9voyez%20vous%20le%20retour%
20de%20Peter%20mc%20Bryan%20%C3%A0%20la%20basse%20?
```

Le script serveur récupère chaque variable et affiche son contenu comme l'illustre la figure 14-9 :



*Figure 14-9. Variables récupérées.*

Nous remarquons que les caractères spéciaux ne s'affichent pas correctement. Cela est dû au fait que le lecteur Flash fonctionne en UTF-8, tandis que PHP fonctionne par défaut en ISO-8859-1.

Nous devons donc décoder la chaîne UTF-8 à l'aide de la méthode PHP `utf8_decode` :

```
<?php

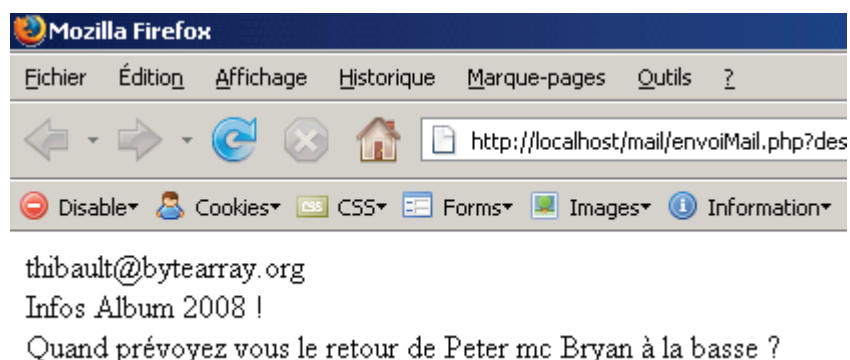
$destinataire = $_GET["destinataire"];
$sujet = $_GET["sujet"];
$message = $_GET["message"];

if ( isset ( $destinataire ) && isset ( $sujet ) && isset ( $message ) )
{
    echo
    utf8_decode($destinataire) . "<br>" . utf8_decode($sujet) . "<br>" . utf8_decode($mes
    sage) ;

    } else echo "Variables non transmises";

?>
```

Si nous testons à nouveau le code précédent, les chaînes sont correctement décodées :



*Figure 14-10. Variables correctement décodées.*

Une fois assuré que les variables passées sont bien réceptionnées, nous pouvons ajouter le code nécessaire pour envoyer l'email.

Pour cela, nous ajoutons l'appel à la fonction PHP `mail` en passant le destinataire, le sujet ainsi que le contenu du message :

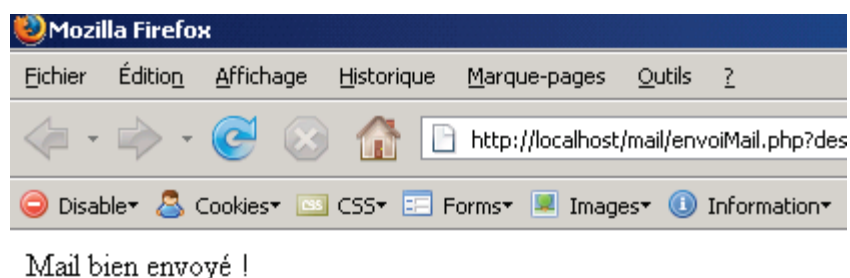
```
<?php
$destinataire = $_GET["destinataire"];
$sujet = $_GET["sujet"];
$message = $_GET["message"];

if ( isset ( $destinataire ) && isset ( $sujet ) && isset ( $message ) )
{
    if ( @mail ( utf8_decode($destinataire), utf8_decode($sujet),
utf8_decode($message) ) ) echo "Mail bien envoyé !";

    else echo "Erreur d'envoi !";
} else echo "Variables non transmises";
?>
```

Nous ajoutons le caractère `@` devant la fonction `mail` afin d'éviter que celle-ci lève une exception PHP en cas de mauvaise configuration du serveur SMTP.

En testant notre application formulaire Flash, nous voyons que le mail est bien envoyé si les variables sont correctement transmises :





*Figure 14-11. Email envoyé.*

Même si le code `ActionScript` précédent fonctionne, il ne fait pas usage de la classe `URLVariables` recommandée dans un contexte d'envoi de variables. Grâce à celle-ci nous allons pouvoir choisir quelle méthode utiliser afin d'envoyer les variables.

Mais avant d'aller plus loin, qu'entendons nous par méthode d'envoi ?

## A retenir

- Nous pouvons envoyer des variables à un script serveur en les ajoutant en fin d'URL.
- Le format d'encodage URL doit alors être respecté.

## La méthode GET ou POST

Lorsque des données sont passées à un script serveur. Celles-ci peuvent être transmises de deux manières différentes. Par l'intermédiaire du tableau GET ou POST.

En utilisant la méthode GET, les variables sont obligatoirement ajoutées en fin d'url. Dans le cas d'un site de vente en ligne, l'adresse suivante permet d'accéder à un article spécifique :

<http://www.funkrecords.com/index.php?rubrique=soul&langue=fr&article=Breakwater>

Un script serveur récupère les variables passées en fin d'URL afin d'afficher le disque correspondant. Lorsque la méthode GET est utilisée, les variables en fin d'URL sont automatiquement accessibles en PHP au sein du tableau associatif `$_GET`.

Il est important de noter que la méthode GET possède une limitation de 1024 caractères, il n'est donc possible de passer un grand volume de données par cette méthode.

Au contraire, lorsque les données sont transmises par la méthode POST, les variables n'apparaissent pas dans l'URL du navigateur. Ainsi, cette méthode ne souffre pas de la limitation de caractères.

Le W3C définit à l'adresse suivante quelques règles concernant l'utilisation des deux méthodes :

<http://www.w3.org/2001/tag/doc/whenToUseGet.html>

Il est conseillé d'utiliser la méthode GET lorsque les données transmises ne sont pas sensibles ou ne sont pas liées à un processus d'écriture.

A l'inverse, la méthode POST est préférée lorsque les données transmises n'ont pas à être exposées et qu'une écriture en base de données intervient par la suite.

L'utilisation d'un objet `URLVariables` va nous permettre de spécifier la méthode à utiliser dans nos échanges entre le lecteur Flash et le script serveur. Au lieu de passer les variables directement après l'url du script distant :

```
var requete:URLRequest = new URLRequest  
("http://localhost/mail/envoiMail.php?destinataire="+destinaireEmail+"& sujet=  
"+sujetEmail+"&message="+messageEmail);
```

Nous allons préférer l'utilisation d'un objet `URLVariables` qui permet de stocker les variables à transmettre. Cet objet est ensuite associé à la propriété `data` de l'objet `URLRequest` utilisé.

Nous pouvons spécifier la méthode à utiliser grâce à la propriété `method` de la classe `URLRequest` et aux constantes de la classe `flash.net.URLRequestMethod`.

Nous modifions le code précédent en utilisant un objet `URLVariables` tout en conservant l'envoi des données avec la méthode GET :

```
package org.bytearray.document  
{  
    import org.bytearray.abstrait.ApplicationDefault;  
    import flash.display.SimpleButton;  
    import flash.text.TextField;  
    import flash.net.navigateToURL;  
    import flash.net.URLRequest;  
    import flash.net.URLVariables;  
    import flash.net.URLRequestMethod;  
    import flash.events.MouseEvent;  
  
    public class Document extends ApplicationDefault  
    {  
  
        public var destinataire:TextField;  
        public var sujet:TextField;  
        public var message:TextField;  
        public var boutonEnvoi:SimpleButton;  
  
        public function Document ()  
        {  
  
            boutonEnvoi.addEventListener ( MouseEvent.CLICK, envoiMail );  
  
        }  
  
        private function envoiMail ( pEvt:MouseEvent ):void
```

```
{  
  
    // création d'un objet URLVariables  
    var variables:URLVariables = new URLVariables();  
  
    // affectation des variables à envoyer coté serveur  
    variables.sujet = sujet.text;  
    variables.destinataire = destinataire.text;  
    variables.message = message.text;  
  
    // création de l'objet URLRequest  
    var requete:URLRequest = new URLRequest  
    ("http://localhost/mail/envoiMail.php");  
  
    // nous passons les variables dans l'url (tableau GET)  
    requete.method = URLRequestMethod.GET;  
  
    // nous associons les variables à l'objet URLRequest  
    requete.data = variables;  
  
    // ouvre une nouvelle fenêtre navigateur et envoi les variables  
    navigateToURL ( requete );  
  
}  
  
}  
  
}
```

Si nous testons à nouveau notre application Flash. Nous remarquons que le script serveur reçoit de la même manière les informations, le mail est bien envoyé.

Afin d'utiliser la méthode POST, nous passons la valeur `URLRequestMethod.POST` à la propriété `method` de l'objet `URLRequest` :

```
// nous passons les variables dans l'url (tableau POST)  
requete.method = URLRequestMethod.POST;
```

Désormais les données seront envoyées au sein du tableau associatif `$_POST`. Nous devons donc impérativement modifier le script serveur afin de réceptionner les variables au sein du tableau correspondant :

```
<?php  
  
$destinataire = $_POST ["destinataire"];  
$sujet = $_POST ["sujet"];  
$message = $_POST ["message"];  
  
if ( isset ( $destinataire ) && isset ( $sujet ) && isset ( $message ) )  
{  
  
    if ( @mail ( utf8_decode($destinataire), utf8_decode($sujet),  
utf8_decode($message) ) ) echo "Mail bien envoyé !";  
  
    else echo "Erreur d'envoi !";  
  
} else echo "Variables non transmises";
```

```
| ?>
```

Lorsque la méthode POST est utilisée, les variables sont automatiquement accessible en PHP au sein du tableau associatif `$_POST`.

---

Attention, si nous tentons d'envoyer des variables à l'aide de la fonction `navigateToURL` à l'aide de la méthode POST depuis l'environnement de test de Flash CS3, celles-ci seront tout de même envoyées par la méthode GET.

Il convient donc de toujours tester au sein du navigateur, une application Flash utilisant la fonction `navigateToURL` et la méthode POST.

---

Nous venons de traiter le moyen le plus simple d'envoyer des variables à un script serveur. Dans la plupart des applications, nous ne souhaitons pas ouvrir de nouvelle fenêtre navigateur lors de la transmission des données. Nous préférons généralement un envoi plus « discret ».

Afin d'envoyer discrètement des données à un script distant, nous préférons l'utilisation de la classe `URLLoader` à la fonction `navigateToURL`.

## A retenir

- Lors de l'envoi de variables par la méthode GET, les variables sont automatiquement ajoutés en fin d'url.
- Lors de l'envoi de variables par la méthode POST, les variables ne sont pas affichées au sein de l'url.
- Il convient d'utiliser la méthode GET pour la lecture de données.
- Il convient d'utiliser la méthode POST pour l'écriture de données.

## Envoyer des variables discrètement

Nous allons modifier notre formulaire développé précédemment afin d'envoyer les informations au script distant, sans ouvrir de nouvelle fenêtre navigateur.

Ainsi l'expérience utilisateur sera plus élégante. Notre application donnera l'impression de fonctionner de manière autonome :

```
| package org.bytearray.document
| {
|     import org.bytearray.abstrait.ApplicationDefault;
```

```
import flash.display.SimpleButton;
import flash.text.TextField;
import flash.net.navigateToURL;
import flash.net.URLRequest;
import flash.net.URLVariables;
import flash.net.URLRequestMethod;
import flash.net.URLLoader;
import flash.events.MouseEvent;

public class Document extends ApplicationDefault
{
    public var destinataire:TextField;
    public var sujet:TextField;
    public var message:TextField;
    public var boutonEnvoi:SimpleButton;
    public var echanges:URLLoader;

    public function Document ()
    {
        echanges = new URLLoader();

        boutonEnvoi.addEventListener ( MouseEvent.CLICK, envoiMail );
    }

    private function envoiMail ( pEvt:MouseEvent ):void
    {
        var variables:URLVariables = new URLVariables();

        // affectation des variables à envoyer coté serveur
        variables.sujet = sujet.text;
        variables.destinataire = destinataire.text;
        variables.message = message.text;

        // création de l'objet URLRequest
        var requete:URLRequest = new URLRequest
("http://localhost/mail/envoiMail.php");

        // nous passons les variables dans l'url (tableau POST)
        requete.method = URLRequestMethod.POST;

        // associe les variables à l'objet URLRequest
        requete.data = variables;

        // envoi les données de manière transparente, sans ouvrir de
nouvelle fenêtre navigateur
        echanges.load ( requete );
    }
}

```

Aussitôt la méthode **load** exécutée, les variables sont transmises au script serveur, l'email est envoyé.

---

En plus de permettre le chargement de données, la méthode `load` permet aussi l'envoi. Cela diffère des précédentes versions d'ActionScript où la classe `LoadVars` possédait une méthode `load` pour le chargement et `send` pour l'envoi.

---

Nous allons continuer l'amélioration de notre formulaire en ajoutant un mécanisme de validation. Ne serait-il pas intéressant de pouvoir indiquer à Flash que le mail a bien été envoyé ?

Pour le moment, notre code n'intègre aucune gestion du retour serveur. C'est ce que nous allons ajouter dans la partie suivante.

## A retenir

- Afin d'envoyer des données de manière transparente, nous utilisons la méthode `load` de l'objet `URLLoader`.
- La méthode `load` permet le chargement de données mais aussi l'envoi.

## Renvoyer des données depuis le serveur

Nous allons à présent nous intéresser à l'envoi de données avec validation. Afin de savoir au sein de Flash si l'envoi du message a bien été effectué, nous devons simplement écouter l'événement `Event.COMPLETE` de l'objet `URLLoader`.

Celui-ci est diffusé automatiquement lorsque le lecteur reçoit des informations de la part du serveur :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.display.SimpleButton;
    import flash.text.TextField;
    import flash.net.navigateToURL;
    import flash.net.URLRequest;
    import flash.net.URLVariables;
    import flash.net.URLRequestMethod;
    import flash.net.URLLoader;
    import flash.events.Event;
    import flash.events.MouseEvent;

    public class Document extends ApplicationDefault
    {
        public var destinataire:TextField;
        public var sujet:TextField;
        public var message:TextField;
        public var boutonEnvoi:SimpleButton;
```

```

public var echanges:URLLoader;

public function Document ()

{

    echanges = new URLLoader();

    echanges.addEventListener ( Event.COMPLETE, retourServeur );

    boutonEnvoi.addEventListener ( MouseEvent.CLICK, envoiMail );

}

private function envoiMail ( pEvt:MouseEvent ):void

{

    var variables:URLVariables = new URLVariables();

    // affectation des variables à envoyer coté serveur
    variables.sujet = sujet.text;
    variables.destinataire = destinataire.text;
    variables.message = message.text;

    // création de l'objet URLRequest
    var requete:URLRequest = new URLRequest
("http://localhost/mail/envoiMail.php");

    // nous passons les variables dans l'url (tableau POST)
    requete.method = URLRequestMethod.POST;

    // associe les variables à l'objet URLRequest
    requete.data = variables;

    // envoi les données de manière transparente, sans ouvrir de
nouvelle fenêtre navigateur
    echanges.load ( requete );

}

private function retourServeur ( pEvt:Event ):void

{

    sujet.text = destinataire.text = message.text = "";

    message.text = pEvt.target.data;

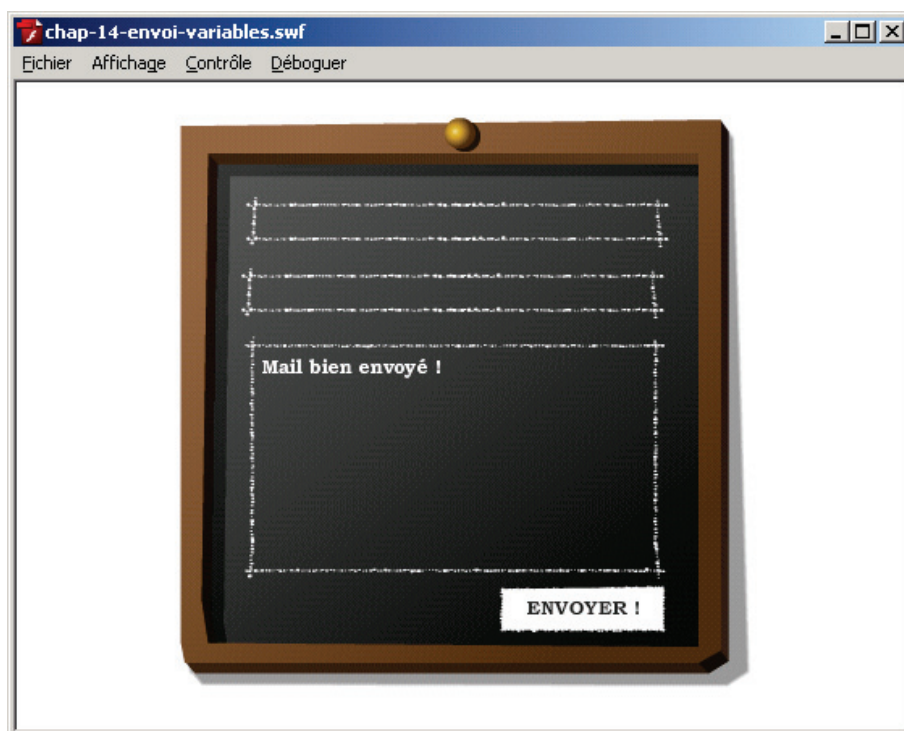
}

}

```

La méthode `retourServeur` est exécutée lorsque le serveur retourne des informations à Flash par l'intermédiaire du mot clé PHP `echo`.

En testant à nouveau l'application, nous obtenons un retour dans le champ message nous indiquant le mail a bien été envoyé, comme l'illustre la figure 14-12 :



*Figure 14-12. Accusé de réception de l'envoi du message.*

Pour des questions de localisation, il n'est pas recommandé de conserver ce script serveur. Car si l'application devait être traduite nous serions obligé de modifier le script serveur.

Afin d'éviter cela, nous allons renvoyer la valeur 1 lorsque le mail est bien envoyé, et 0 le cas échéant. Nous renvoyons la valeur 2 lorsque les variables ne sont pas bien réceptionnées :

```
<?php
$destinataire = $_POST ["destinataire"];
$sujet = $_POST ["sujet"];
$message = $_POST ["message"];

if ( isset ( $destinataire ) && isset ( $sujet ) && isset ( $message ) )
{
    if ( @mail ( utf8_decode($destinataire), utf8_decode($sujet),
utf8_decode($message) ) ) echo "resultat=1";

    else echo "resultat=0";
} else echo "resultat=2";
?>
```



Nous décidons ainsi côté client quel message afficher. Nous renvoyons donc une chaîne encodée URL qui devra être décodée coté client.

Pour cela nous demandons à l'objet `URLLoader` de décoder toutes les chaînes reçues afin de pouvoir facilement extraire le contenu de la variable `resultat`. Nous modifions la méthode `retourServeur` afin d'afficher le message approprié :

```
package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefault;
    import flash.display.SimpleButton;
    import flash.text.TextField;
    import flash.net.navigateToURL;
    import flash.net.URLRequest;
    import flash.net.URLVariables;
    import flash.net.URLRequestMethod;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.events.Event;
    import flash.events.MouseEvent;
    import flash.events.HTTPStatusEvent;
    import flash.events.IOErrorEvent;

    public class Document extends ApplicationDefault

    {

        public var destinataire:TextField;
        public var sujet:TextField;
        public var message:TextField;
        public var boutonEnvoi:SimpleButton;
        public var echanges:URLLoader;

        public function Document ()

        {

            echanges = new URLLoader();

            echanges.dataFormat = URLLoaderDataFormat.VARIABLES;

            echanges.addEventListener ( Event.COMPLETE, retourServeur );
            echanges.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );
            echanges.addEventListener ( HTTPStatusEvent.HTTP_STATUS,
statutHTTP );

            boutonEnvoi.addEventListener ( MouseEvent.CLICK, envoiMail );

        }

        private function envoiMail ( pEvt:MouseEvent ):void

        {

            var variables:URLVariables = new URLVariables();
```

```

        // affectation des variables à envoyer coté serveur
        variables.sujet = sujet.text;
        variables.destinataire = destinataire.text;
        variables.message = message.text;

        // création de l'objet URLRequest
        var requete:URLRequest = new URLRequest
("http://localhost/mail/envoiMail.php");

        // nous passons les variables dans l'url (tableau POST )
        requete.method = URLRequestMethod.POST;

        // associe les variables à l'objet URLRequest
        requete.data = variables;

        // envoi les données de manière transparente, sans ouvrir de
nouvelle fenêtre navigateur
        echanges.load ( requete );

    }

    private function retourServeur ( pEvt:Event ):void

    {

        sujet.text = destinataire.text = message.text = "";

        var messageRetour:String;

        if ( Number ( pEvt.target.data.resultat ) == 1 ) messageRetour =
"Mail bien envoyé !";

        else if ( Number ( pEvt.target.data.resultat ) == 2 )
messageRetour = "Erreur de transmission des données !";

        else messageRetour = "Erreur d'envoi du message !";

        message.text = messageRetour;

    }

    private function statutHTTP ( pEvt:HTTPStatusEvent ):void

    {

        trace( "Code HTTP : " + pEvt.status );

    }

    private function erreurChargement ( pEvt:IOErrorEvent ):void

    {

        trace("Erreur de chargement !");

    }

}

}

```

Si nous n'envoyons pas les variables au script serveur ou que la fonction `mail` échoue nous obtenons un message approprié dans notre application Flash. Dans le code suivant, nous n'affectons aucune variable à l'objet `URLVariables` :

```
private function envoiMail ( pEvt:MouseEvent ):void
{
    var variables:URLVariables = new URLVariables();

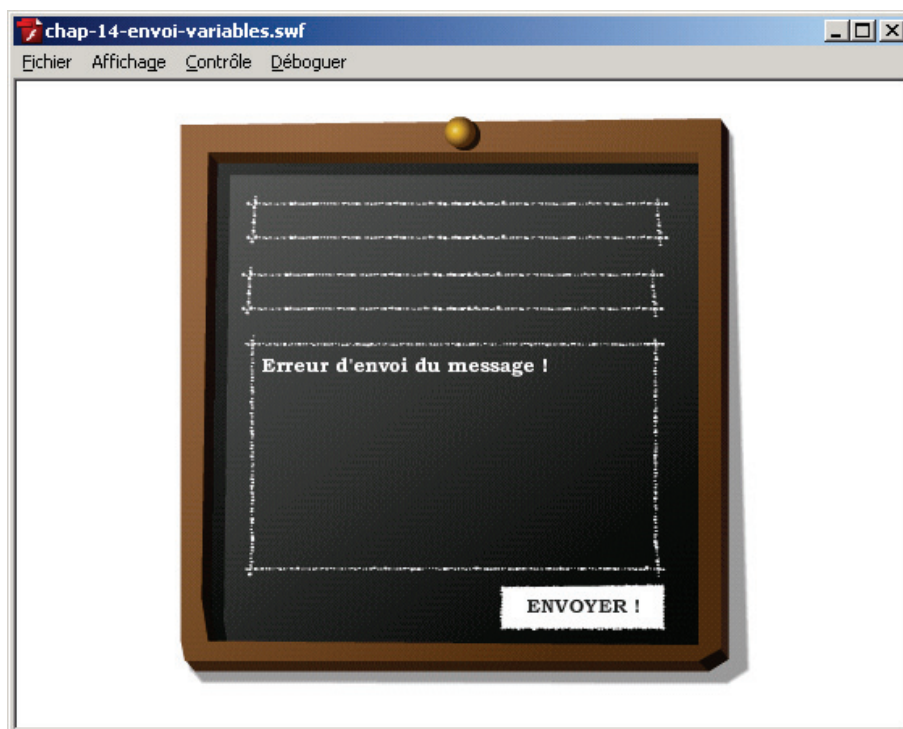
    // création de l'objet URLRequest
    var requete:URLRequest = new URLRequest
    ("http://localhost/mail/envoiMail.php");

    // nous passons les variables dans l'url (tableau POST )
    requete.method = URLRequestMethod.POST;

    // associe les variables à l'objet URLRequest
    requete.data = variables;

    // envoi les données de manière transparente, sans ouvrir de nouvelle
    fenêtre navigateur
    echanges.load ( requete );
}
```

L'email ne peut être envoyé, la figure 14-13 illustre le message affiché par l'application :



*Figure 14-13. Echec d'envoi de l'email.*

Nous pourrions aller plus loin dans cet exemple et enregistrer l'utilisateur dans une base de données et renvoyer d'autres types d'informations.

Nous reviendrons en détail sur l'envoi et la réception de données issues d'une base de données au cours du chapitre 19 intitulé *Remoting*.

## A retenir

- Afin de récupérer les données issues du serveur nous écoutons l'événement `Event.COMPLETE` de l'objet `URLLoader`.
- Afin de retourner des données à Flash, nous devons utiliser le mot clé PHP `echo`.

## Aller plus loin

Souvenez-vous, au cours du chapitre 2, nous avons découvert la notion d'expressions régulières permettant d'effectuer des manipulations complexes sur des chaînes de caractères.

Afin d'optimiser notre application nous allons intégrer une vérification de l'email par expression régulière.

Nous allons donc créer une classe `OutilsFormulaire` globale à tous nos projets, intégrant une méthode statique `verifieEmail`. Celle-ci renverra `true` si l'email est correcte ou `false` le cas échéant.

Rappelez-vous, au cours du chapitre 12 intitulé *Programmation Bitmap*, nous avons créé un répertoire global de classes nommé `classes_as3`. Au sein du répertoire `utils` nous créons la classe `OutilsFormulaire` suivante :

```
package org.bytearray.utils
{
    public class FormulaireUtils
    {
        private static var emailModele:RegExp = /^[a-z0-9][-._a-z0-9]*@([a-z0-9]
        [-_a-z0-9]*\.)+[a-z]{2,6}$/

        public static function verifieEmail ( pEmail:String ):Boolean
        {
            var resultat:Array = pEmail.match( FormulaireUtils.emailModele
        );

            return resultat != null;
        }
    }
}
```

```
    }
  }
}
```

Une fois définie, nous pouvons l'utiliser dans notre application en l'important :

```
import org.bytearray.ouutils.FormulaireOutils;
```

Puis nous modifions la méthode `envoiMail` afin d'intégrer une vérification de l'email :

```
private function envoiMail ( pEvt:MouseEvent ):void
{
    if ( FormulaireOutils.verifieEmail ( destinataire.text ) )
    {
        var variables:URLVariables = new URLVariables();

        // affectation des variables à envoyer coté serveur
        variables.sujet = sujet.text;
        variables.destinataire = destinataire.text;
        variables.message = message.text;

        // création de l'objet URLRequest
        var requete:URLRequest = new URLRequest
        ("http://localhost/mail/envoiMail.php");

        // nous passons les variables dans l'url (tableau POST )
        requete.method = URLRequestMethod.POST;

        // associe les variables à l'objet URLRequest
        requete.data = variables;

        // envoi les données de manière transparente, sans ouvrir de nouvelle
        fenêtre navigateur
        echanges.load ( requete );

    } else destinataire.text = "Email non valide !";
}
```

Ainsi, toutes nos applications pourront utiliser la classe `FormulaireOutils` afin de vérifier la validité d'un email. Bien entendu, nous pouvons ajouter d'autres méthodes pratiques à celle-ci.

Nous retrouvons ici l'intérêt d'une méthode statique, pouvant être appelée directement sur le constructeur de la classe, sans avoir à instancier un objet `FormulaireOutils`.

## Envoyer un flux binaire

Rendez vous au chapitre 20 intitulé `ByteArray` pour découvrir comment transmettre un flux binaire grâce à la classe `URLLoader`.

## Télécharger un fichier

Nous avons vu jusqu'à présent comment charger ou envoyer des variables à partir du lecteur Flash. Certaines applications peuvent néanmoins nécessiter un téléchargement ou un envoi de fichiers entre le serveur et l'ordinateur de l'utilisateur.

La classe `flash.net.FileReference` permet de télécharger n'importe quel fichier grâce à la méthode `download`.

Imaginons que nous devons télécharger un fichier zip depuis une URL spécifique telle :

<http://www.monserveur.org/archive.zip>

Au sein d'un nouveau document Flash CS3 nous associons la classe de document suivante :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault
    {
        public function Document ()
        {

        }

    }
}
```

Puis nous créons un objet `FileReference` :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.FileReference;

    public class Document extends ApplicationDefault
    {
        private var telechargeur:FileReference;

        public function Document ()
        {
```

```

        telechargeur = new FileReference();
    }
}
}

```

Afin de gérer les différentes étapes liées au téléchargement de données la classe `FileReference` diffuse différents événements dont voici le détail :

- `Event.CANCEL` : diffusé lorsqu'un envoi ou téléchargement est annulé par la fenêtre de parcours.
- `Event.COMPLETE` : diffusé lorsqu'un envoi ou un téléchargement a réussi.
- `HTTPStatusEvent.HTTP_STATUS` : diffusé lorsqu'un envoi ou chargement échoue.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque l'envoi ou la réception des données échoue.
- `Event.OPEN` : diffusé lorsque l'envoi ou la réception des données commence.
- `ProgressEvent.PROGRESS` : diffusé lorsque l'envoi ou le chargement est en cours.
- `SecurityErrorEvent.SECURITY_ERROR` : diffusé lorsque le lecteur tente d'envoyer ou de charger des données depuis un domaine non autorisé.
- `Event.SELECT` : diffusé lorsque le bouton OK de la boîte de dialogue de recherches de fichiers est relâché.
- `DataEvent.UPLOAD_COMPLETE_DATA` : diffusé lorsqu'une opération d'envoi ou de chargement est terminée et que le serveur renvoie des données.

Afin de télécharger un fichier nous utilisons la méthode `download` dont voici la signature :

```

public function download(request:URLRequest, defaultFileName:String = null):void

```

Celle-ci requiert deux paramètres dont voici le détail :

- `request` : un objet `URLRequest` contenant l'adresse de l'élément à télécharger.
- `defaultFileName` : ce paramètre permet de spécifier le nom de l'élément téléchargé au sein de la boîte de dialogue.

Dans le code suivant, nous téléchargeons un fichier distant :

```

package org.bytearray.document

```

```

{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;

    public class Document extends ApplicationDefault
    {
        private var telechargeur:FileReference;
        private var lienFichier:String = "http://alivepdf.bytearray.org/wp-
content/images/logo_small.jpg";
        private var requete:URLRequest = new URLRequest( lienFichier );
        public var boutonTelecharger:SimpLeButton;

        public function Document ()
        {
            // création d'un objet FileReference
            telechargeur = new FileReference();

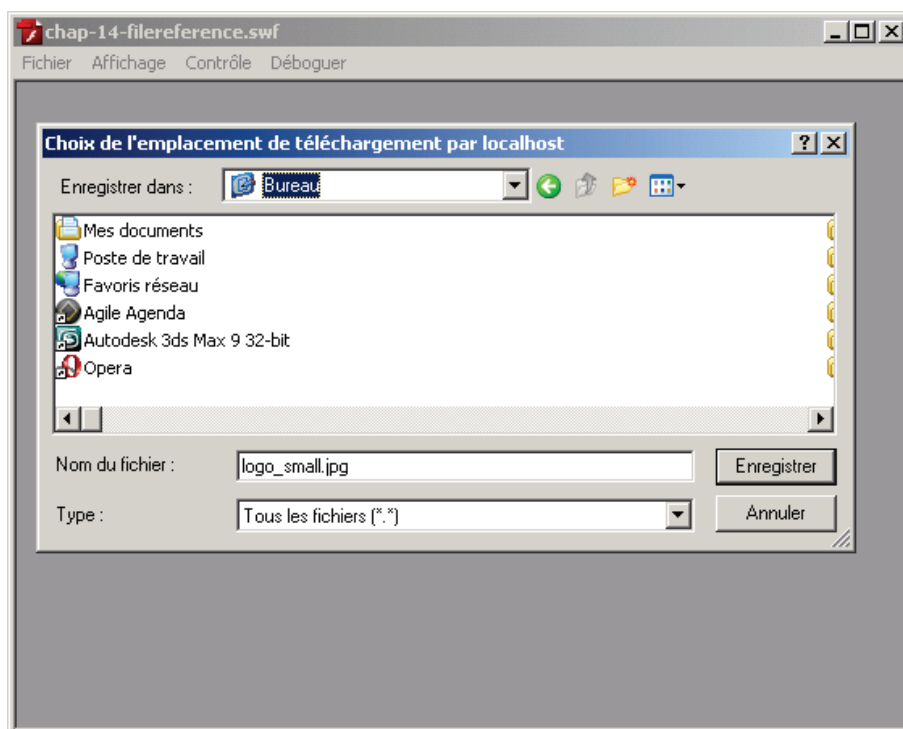
            boutonTelecharger.addEventListener ( MouseEvent.CLICK,
            telechargeFichier );
        }

        private function telechargeFichier ( pEvt:MouseEvent ):void
        {
            // téléchargement du fichier distant
            telechargeur.download ( requete );
        }
    }
}

```

Lorsque nous cliquons sur le bouton `boutonTelecharger` une fenêtre de téléchargement s'ouvre comme l'illustre la figure 14-14 :





*Figure 14-14. Boite de dialogue.*

Une fois l'emplacement sélectionné, nous cliquons sur le bouton **Enregistrer**. Le fichier est alors sauvé en local sur l'ordinateur exécutant l'application. Bien que nous n'ayons pas spécifié le nom du fichier, le lecteur Flash extrait automatiquement celui-ci à partir de son URL.

Attention, lorsque cette boite de dialogue apparaît, la tentative d'accès au fichier distant n'a pas encore démarré. Si nous modifions l'adresse du fichier distant par l'adresse suivante :

```
private var lienFichier:String =
    "http://www.monserveurInexistant.org/logo_small.jpg";
```

Le lecteur affiche tout de même la boite de dialogue ainsi que le nom du fichier.

Afin de gérer l'état du transfert nous écoutons les principaux événements :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefaut;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;
```

```
import flash.events.HTTPStatusEvent;
import flash.events.ProgressEvent;
import flash.events.IOErrorEvent;

public class Document extends ApplicationDefault
{
    private var telechargeur:FileReference;
    private var lienFichier:String = "http://alivepdf.bytearray.org/wp-
content/images/logo_small.jpg";
    private var requete:URLRequest = new URLRequest( lienFichier );
    public var boutonTelecharger:SimpleButton;

    public function Document ()
    {
        // création d'un objet FileReference
        telechargeur = new FileReference();

        telechargeur.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
        telechargeur.addEventListener ( Event.COMPLETE, chargementTermine
);
        telechargeur.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );

        boutonTelecharger.addEventListener ( MouseEvent.CLICK,
telechargeFichier );
    }

    private function chargementEnCours ( pEvt:ProgressEvent ):void
    {
        trace( "chargement en cours : " + pEvt.bytesLoaded /
pEvt.bytesTotal );
    }

    private function chargementTermine ( pEvt:Event ):void
    {
        trace( "chargement terminé" );
    }

    private function erreurChargement ( pEvt:IOErrorEvent ):void
    {
        trace( "erreur de chargement du fichier !" );
    }

    private function telechargeFichier ( pEvt:MouseEvent ):void
    {

```

```
        // téléchargement du fichier distant
        telechargeur.download ( requete );

    }

}

}
```

La méthode `download` n'accepte qu'un seul fichier à télécharger. Il est donc impossible de télécharger plusieurs fichiers en même temps à l'aide de la classe `FileReference`.

Il est important de noter que dans un contexte inter domaines, le téléchargement de fichiers est interdit si le domaine distant n'est pas autorisé.

Ainsi, si nous publions l'application actuelle sur un serveur distant, l'événement `SecurityErrorEvent.SECURITY_ERROR` est diffusé affichant le message suivant :

```
Error #2048: Violation de la sécurité Sandbox :
http://monserveur.fr/as3/chap-14-filereference.swf ne peut pas charger de
données à partir de http://alivepdf.bytearray.org/wp-
content/images/logo_small.jpg.
```

Afin de pouvoir charger le fichier distant nous devons utiliser un *fichier de régulation* ou un *proxy*.

Nous allons utiliser dans le code suivant un *proxy* afin de feindre le lecteur Flash :

```
package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefaut;
    import flash.net.URLRequestMethod;
    import flash.net.URLVariables;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.events.HTTPStatusEvent;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;
    import flash.events.SecurityErrorEvent;

    public class Document extends ApplicationDefaut

    {

        private var telechargeur:FileReference;
        private var lienFichier:String = "proxy.php";
        private var requete:URLRequest = new URLRequest( lienFichier );
        public var boutonTelecharger:SimpleButton;
```

```

        public function Document ()
        {
            // création d'un objet FileReference
            telechargeur = new FileReference();

            // création d'un objet URLVariables pour passer le chemin du
            // fichier à charger au proxy
            var variables:URLVariables = new URLVariables();

            // spécification du chemin
            variables.chemin = "http://alivepdf.bytearray.org/wp-
            content/images/logo_small.jpg";

            // affectation des variables et de la méthode utilisée
            requete.data = variables;
            requete.method = URLRequestMethod.POST;

            telechargeur.addEventListener ( ProgressEvent.PROGRESS,
            chargementEnCours );
            telechargeur.addEventListener ( Event.COMPLETE, chargementTermine
            );
            telechargeur.addEventListener ( IOErrorEvent.IO_ERROR,
            erreurChargement );

            boutonTelecharger.addEventListener ( MouseEvent.CLICK,
            telechargeFichier );
        }

        private function chargementEnCours ( pEvt:ProgressEvent ):void
        {
            trace( "chargement en cours : " + pEvt.bytesLoaded /
            pEvt.bytesTotal );
        }

        private function chargementTermine ( pEvt:Event ):void
        {
            trace( "chargement terminé" );
        }

        private function erreurChargement ( pEvt:IOErrorEvent ):void
        {
            trace( "erreur de chargement du fichier !" );
        }

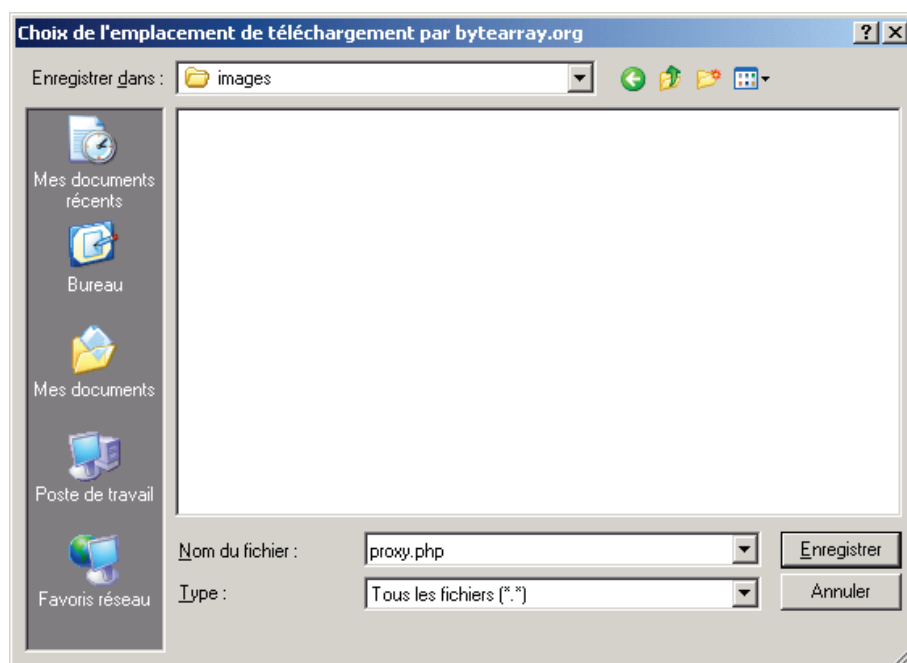
        private function telechargeFichier ( pEvt:MouseEvent ):void
        {
            // téléchargement du fichier distant
            telechargeur.download ( requete );
        }
    
```

```

    }
}
}

```

Une fois l'application publiée, l'appel de la méthode `download` ne lève plus d'exception. La boîte de dialogue s'ouvre et propose comme nom de fichier le nom du fichier proxy utilisé comme l'illustre la figure 14-15 :



*Figure 14-15. Nom d'enregistrement du fichier.*

Comme nous l'avons vu précédemment, le lecteur devine automatiquement le nom du fichier téléchargé depuis l'URL du fichier. Dans notre cas, ce dernier considère que le fichier à charger est le proxy et propose donc comme nom de fichier `proxy.php`.

Afin de proposer le nom réel du fichier chargé, nous allons utiliser le deuxième paramètre `defaultFileName` de la méthode `download` :

```

package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefaut;
    import flash.net.URLRequestMethod;
    import flash.net.URLVariables;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;

```

```

import flash.events.HTTPStatusEvent;
import flash.events.ProgressEvent;
import flash.events.IOErrorEvent;
import flash.events.SecurityErrorEvent;

public class Document extends ApplicationDefault
{
    private var telechargeur:FileReference;
    private var lienFichier:String = "proxy.php";
    private var requete:URLRequest = new URLRequest( lienFichier );
    private var urlFichier:String;
    public var boutonTelecharger:SimpleButton;

    public function Document ()
    {
        // création d'un objet FileReference
        telechargeur = new FileReference();

        // création d'un objet URLVariables pour passer le chemin du
        fichier à charger au proxy
        var variables:URLVariables = new URLVariables();

        // url du fichier distant à charger par le proxy
        urlFichier = "http://alivepdf.bytearray.org/wp-
content/images/logo_small.jpg";

        // spécification du chemin
        variables.chemin = urlFichier;

        // affectation des variables et de la méthode utilisée
        requete.data = variables;
        requete.method = URLRequestMethod.POST;

        telechargeur.addEventListener ( ProgressEvent.PROGRESS,
        chargementEnCours );
        telechargeur.addEventListener ( Event.COMPLETE, chargementTermine
    );
        telechargeur.addEventListener ( IOErrorEvent.IO_ERROR,
        erreurChargement );

        boutonTelecharger.addEventListener ( MouseEvent.CLICK,
        telechargeFichier );
    }

    private function chargementEnCours ( pEvt:ProgressEvent ):void
    {
        trace( "chargement en cours : " + pEvt.bytesLoaded /
        pEvt.bytesTotal );
    }

    private function chargementTermine ( pEvt:Event ):void
    {

```

```

        trace( "chargement terminé" );
    }

    private function erreurChargement ( pEvt:IOErrorEvent ):void
    {
        trace( "erreur de chargement du fichier !" );
    }

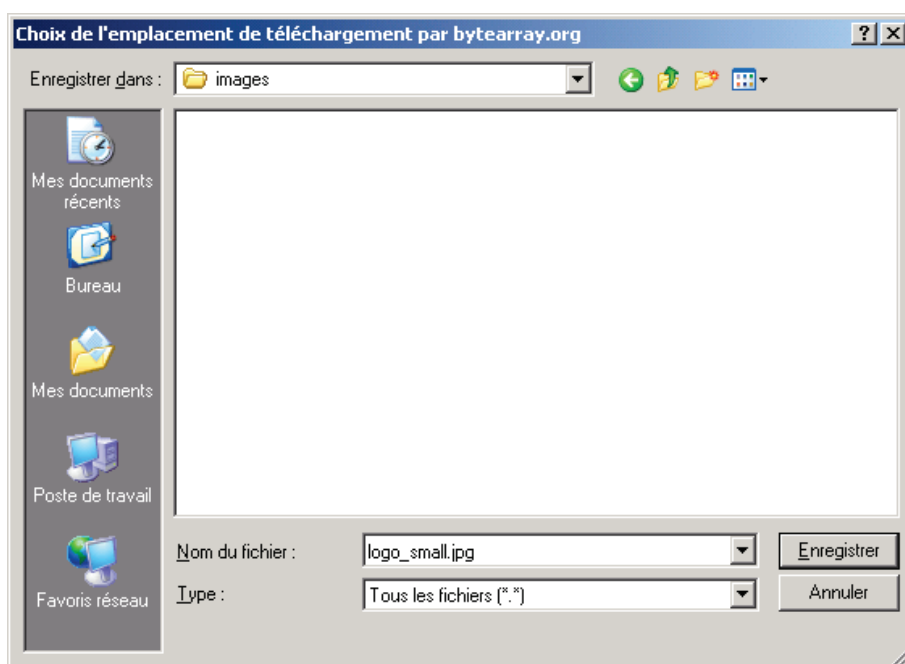
    private function telechargeFichier ( pEvt:MouseEvent ):void
    {
        // expression régulière
        var modele:RegExp = /[A-Za-z0-9_]*\.D{3}$/

        // extrait le nom du fichier de l'url
        var resultat:Array = urlFichier.match ( modele );

        // téléchargement du fichier distant
        if ( resultat != null ) telechargeur.download ( requete,
        resultat[0] );
    }
}

```

Le nom du fichier est extrait manuellement puis passé en deuxième paramètre de la méthode `download`. Lors du clic sur le bouton `boutonTelecharger` la boîte de dialogue s'ouvre en proposant le bon nom de fichier :



*Figure 14-16. Nom d'enregistrement du fichier.*

Il est donc possible de rendre totalement transparent l'intervention d'un *proxy* dans le téléchargement de notre fichier distant. Lorsque nous cliquons sur le bouton `Enregistrer`, le fichier est téléchargé et enregistré.

Nous allons nous attarder à présent sur l'envoi de fichiers par la classe `flash.net.FileReference`.

## A retenir

- Afin de télécharger un fichier depuis le lecteur Flash nous utilisons la méthode `download` de la classe `FileReference`.
- L'appel de la méthode `download` entraîne l'ouverture d'une boîte de dialogue permettant de sauver le fichier sur l'ordinateur de l'utilisateur.

## Publier un fichier

A l'inverse si nous souhaitons mettre en ligne un fichier sélectionné depuis l'ordinateur de l'utilisateur, nous pouvons utiliser la méthode `upload` de la classe `FileReference`.

Dans un nouveau document Flash CS3, nous associons la classe de document suivante :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;

    public class Document extends ApplicationDefault
    {
        private var envoyeur:FileReference;
        private var lienScript:String =
"http://localhost/envoi/envoiFichier.php";
        private var requete:URLRequest = new URLRequest( lienScript );
        public var boutonEnvoyer:SimpleButton;

        public function Document ()
        {
            // création d'un objet FileReference
        }
    }
}
```



```

        envoyeur = new FileReference();

        envoyeur.addEventListener ( ProgressEvent.PROGRESS, envoiEnCours
    );

        envoyeur.addEventListener ( Event.COMPLETE, envoiTermine );
        envoyeur.addEventListener ( IOErrorEvent.IO_ERROR, erreurEnvoi );

        boutonEnvoyer.addEventListener ( MouseEvent.CLICK,
parcoursFichiers );

    }

    private function envoiEnCours ( pEvt:ProgressEvent ):void
    {

        trace( "envoi en cours : " + pEvt.bytesLoaded / pEvt.bytesTotal
    );

    }

    private function envoiTermine ( pEvt:Event ):void
    {

        trace( "envoi terminé" );

    }

    private function erreurEnvoi ( pEvt:IOErrorEvent ):void
    {

        trace( "erreur d'envoi du fichier !" );

    }

    private function parcoursFichiers ( pEvt:MouseEvent ):void
    {

        // ouvre la boite de dialogue permettant de parcourir les
fichiers
        envoyeur.browse ();

    }

}

}

```

Au sein de l'application, un bouton `boutonEnvoyer` déclenche l'ouverture de la boite de dialogue permettant de parcourir les fichiers à transférer à l'aide de la méthode `browse` de l'objet `FileReference`.

La propriété `lienScript` pointe vers un script serveur permettant l'enregistrement du fichier transféré sur le serveur. Voici le code PHP permettant de sauver le fichier transféré :

```
<?php
$fichier = $_FILES["Filedata"];
if ( isset ( $fichier ) )
{
    $nomFichier = $fichier['name'];
    move_uploaded_file ( $fichier['tmp_name'], "monRepertoire/".$nomFichier);
}
?>
```

Le lecteur Flash place le fichier transféré au sein du tableau `$_FILES` et de la propriété `Filedata`.

Lors de la gestion d'envoi de fichiers par `FileReference` il convient d'écouter l'événement `Event.SELECT` diffusé lorsque l'utilisateur a sélectionné un fichier. Ainsi nous pouvons récupérer différentes informations liées au fichier sélectionné :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;

    public class Document extends ApplicationDefault
    {
        private var envoyeur:FileReference;
        private var lienScript:String =
"http://localhost/envoi/envoiFichier.php";
        private var requete:URLRequest = new URLRequest( lienScript );
        public var boutonEnvoyer:SimpleButton;

        public function Document ()
        {
            // création d'un objet FileReference
            envoyeur = new FileReference();

            envoyeur.addEventListener ( Event.SELECT, selectionFichier );
            envoyeur.addEventListener ( ProgressEvent.PROGRESS, envoiEnCours );
        };

        envoyeur.addEventListener ( Event.COMPLETE, envoiTermine );
        envoyeur.addEventListener ( IOErrorEvent.IO_ERROR, erreurEnvoi );
    }
}
```

```
        boutonEnvoyer.addEventListener ( MouseEvent.CLICK,
parcoursFichiers );

    }

    private function selectionFichier ( pEvt:Event ):void
    {

        var fichier:FileReference = FileReference ( pEvt.target );

        // affiche : Tue Oct 23 19:59:27 GMT+0200 2007
        trace( fichier.creationDate );

        // affiche : Interview.doc
        trace( fichier.name );

        // affiche : 37888
        trace( fichier.size );

        // affiche : .doc
        trace( fichier.type );

    }

    private function envoiEnCours ( pEvt:ProgressEvent ):void
    {

        trace( "envoi en cours : " + pEvt.bytesLoaded / pEvt.bytesTotal
);

    }

    private function envoiTermine ( pEvt:Event ):void
    {

        trace( "envoi terminé" );

    }

    private function erreurEnvoi ( pEvt:IOErrorEvent ):void
    {

        trace( "erreur d'envoi du fichier !" );

    }

    private function parcoursFichiers ( pEvt:MouseEvent ):void
    {

        // ouvre la boite de dialogue permettant de parcourir les
fichiers
        envoyeur.browse ();

    }

}
```

```
| }
```

Voici le détail de chacune des propriétés définies par la classe `FileReference` :

- `creationDate` : date de création du fichier.
- `creator` : type de créateur Macintosh du fichier.
- `modificationDate` : date de dernière modification du fichier.
- `name` : nom du fichier.
- `size` : taille du fichier en octets.
- `type` : extension du fichier.

Une fois l'objet sélectionné, nous appelons la méthode `upload` sur l'objet `FileReference` :

```
private function selectionFichier ( pEvt:Event ):void
{
    var fichier:FileReference = FileReference ( pEvt.target );

    // affiche : Tue Oct 23 19:59:27 GMT+0200 2007
    trace( fichier.creationDate );

    // affiche : Interview.doc
    trace( fichier.name );

    // affiche : 37888
    trace( fichier.size );

    // affiche : .doc
    trace( fichier.type );

    // envoi du fichier
    fichier.upload ( requete );
}
```

Comme l'illustre la figure 14-17, vous remarquerez qu'il est impossible de sélectionner plusieurs fichiers au sein de la boîte de dialogue ouverte par la méthode `browse` de l'objet `FileReference` :



*Figure 14-17. Selection multiple impossible.*

Afin de pouvoir sélectionner plusieurs fichiers nous devons obligatoirement utiliser la classe `flash.net.FileReferenceList`.

## A retenir

- Afin de sélectionner un fichier à transférer sur un serveur depuis le lecteur Flash nous utilisons d'abord la méthode `browse` de l'objet `FileReference`.
- Une fois le fichier sélectionné, nous appelons la méthode `upload` en spécifiant l'URL du script serveur permettant l'enregistrement du fichier.

## Publier plusieurs fichiers

Afin de pouvoir envoyer plusieurs fichiers en même temps, nous modifions la classe du document afin de créer un objet `FileReferenceList` :

```
package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.FileReferenceList;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;

    public class Document extends ApplicationDefault
    {
        private var envoyeurMultiple:FileReferenceList;
```

```

        private var lienScript:String =
"http://localhost/envoi/envoiFichier.php";
        private var requete:URLRequest = new URLRequest( lienScript );
        public var boutonEnvoyer:SimpleButton;

        public function Document ()

        {

            // création d'un objet FileReference
            envoyeurMultiple = new FileReferenceList();

            envoyeurMultiple.addEventListener ( Event.SELECT,
selectionFichier );
            envoyeurMultiple.addEventListener ( ProgressEvent.PROGRESS,
envoiEnCours );
            envoyeurMultiple.addEventListener ( Event.COMPLETE, envoiTermine
);
            envoyeurMultiple.addEventListener ( IOErrorEvent.IO_ERROR,
erreurEnvoi );

            boutonEnvoyer.addEventListener ( MouseEvent.CLICK,
parcoursFichiers );

        }

        private function selectionFichier ( pEvt:Event ):void

        {

            // référence le tableau contenant chaque objet FileReference
            var listeFichiers:Array = pEvt.target.fileList;

            // affiche : n fichiers sélectionnés
            trace( listeFichiers.length + " fichier(s) sélectionnés");

        }

        private function envoiEnCours ( pEvt:ProgressEvent ):void

        {

            trace( "envoi en cours : " + pEvt.bytesLoaded / pEvt.bytesTotal
);

        }

        private function envoiTermine ( pEvt:Event ):void

        {

            trace( "envoi terminé" );

        }

        private function erreurEnvoi ( pEvt:IOErrorEvent ):void

        {

            trace( "erreur d'envoi du fichier !" );

        }

    }

```

```

private function parcoursFichiers ( pEvt:MouseEvent ):void
{
    // ouvre la boite de dialogue permettant de parcourir les
    fichiers
    envoyeurMultiple.browse ();
}
}
}

```

La boîte de dialogue ouverte par la méthode `browse` de l'objet `FileReferenceList` permet la sélection multiple, comme l'indique la figure 14-18 :



*Figure 14-18. Sélection multiple possible.*

En réalité lorsque nous sélectionnons plusieurs fichiers à travers la boîte de dialogue et que nous validons la sélection. La classe `FileReferenceList` crée un interne un objet `FileReference` associé à chaque fichier référence au sein de la propriété `fileList` de l'objet `FileReferenceList`.

Nous devons donc manuellement parcourir ce tableau interne contenant les objets `FileReference` et appeler la méthode `upload` sur chacun d'entre eux.

Il est important de signaler que l'objet `FileReference` ne diffuse que les événements `Event.SELECT` et `Event.CANCEL` liés à la sélection des fichiers. La gestion du chargement se fait par l'intermédiaire des objets `FileReference` créés en interne par l'objet `FileReferenceList` :

```
package org.bytearray.document
```

```

{
    import org.bytearray.abstrait.ApplicationDefault;
    import flash.net.FileReferenceList;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;

    public class Document extends ApplicationDefault
    {
        private var envoyeurMultiple:FileReferenceList;
        private var lienScript:String =
"http://localhost/envoi/envoiFichier.php";
        private var requete:URLRequest = new URLRequest( lienScript );
        public var boutonEnvoyer:SimpleButton;

        public function Document ()
        {
            // création d'un objet FileReference
            envoyeurMultiple = new FileReferenceList();

            envoyeurMultiple.addEventListener ( Event.SELECT,
selectionFichier );

            boutonEnvoyer.addEventListener ( MouseEvent.CLICK,
parcoursFichiers );
        }

        private function selectionFichier ( pEvt:Event ):void
        {
            // référence le tableau contenant chaque objet FileReference
            var listeFichiers:Array = pEvt.target.fileList;

            // longueur du tableau
            var lng:int = listeFichiers.length;

            for ( var i:int = 0; i< lng; i++ )
            {
                var fichier:FileReference = listeFichiers[i];

                fichier.addEventListener ( ProgressEvent.PROGRESS,
envoiEnCours );
                fichier.addEventListener ( Event.COMPLETE, envoiTermine );
                fichier.addEventListener ( IOErrorEvent.IO_ERROR,
erreurEnvoi );

                fichier.upload ( requete );
            }
        }
    }
}

```



```

    }

    }

    private function envoiEnCours ( pEvt:ProgressEvent ):void
    {
        trace( "envoi en cours : " + pEvt.bytesLoaded / pEvt.bytesTotal
    );
    }

    private function envoiTermine ( pEvt:Event ):void
    {
        trace( "envoi terminé" );
    }

    private function erreurEnvoi ( pEvt:IOErrorEvent ):void
    {
        trace( "erreur d'envoi du fichier !" );
    }

    private function parcoursFichiers ( pEvt:MouseEvent ):void
    {
        // ouvre la boite de dialogue permettant de parcourir les
        fichiers
        envoyeurMultiple.browse ();
    }
}
}

```

En testant notre application, les fichiers sont bien transférés sur le serveur.

Nous allons ajouter à présent une jauge de chargement indiquant l'état du transfert. Pour cela nous créons un clip de forme rectangulaire en bibliothèque auquel nous associons une classe **Prechargeur**.

```

package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefaut;
    import flash.net.FileReferenceList;
    import flash.net.FileReference;
    import flash.net.URLRequest;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.events.Event;

```

```

import flash.events.ProgressEvent;
import flash.events.IOErrorEvent;

public class Document extends ApplicationDefault
{
    private var envoyeurMultiple:FileReferenceList;
    private var lienScript:String =
"http://localhost/envoi/envoiFichier.php";
    private var requete:URLRequest = new URLRequest( lienScript );
    public var boutonEnvoyer:SimpleButton;
    private var prechargeur:Prechargeur;

    public function Document ()
    {
        // création d'un objet FileReference
        envoyeurMultiple = new FileReferenceList();

        prechargeur = new Prechargeur();

        envoyeurMultiple.addEventListener ( Event.SELECT,
selectionFichier );

        boutonEnvoyer.addEventListener ( MouseEvent.CLICK,
parcoursFichiers );
    }

    private function selectionFichier ( pEvt:Event ):void
    {
        // référence le tableau contenant chaque objet FileReference
        var listeFichiers:Array = pEvt.target.fileList;

        // longueur du tableau
        var lng:int = listeFichiers.length;

        for ( var i:int = 0; i< lng; i++ )
        {
            var fichier:FileReference = listeFichiers[i];

            fichier.addEventListener ( ProgressEvent.PROGRESS,
envoiEnCours );
            fichier.addEventListener ( Event.COMPLETE, envoiTermine );
            fichier.addEventListener ( IOErrorEvent.IO_ERROR,
erreurEnvoi );

            fichier.upload ( requete );
        }

        // ajout du préchargeur à l'affichage
        if ( !contains ( prechargeur ) ) addChild ( prechargeur );
    }
}

```

```

        private function envoiEnCours ( pEvt:ProgressEvent ):void
        {
            // ajuste la taille de la jauge par rapport à l'état du
transfert
            prechargeur.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;
        }

        private function envoiTermine ( pEvt:Event ):void
        {
            trace( "envoi terminé" );
        }

        private function erreurEnvoi ( pEvt:IOErrorEvent ):void
        {
            trace( "erreur d'envoi du fichier !" );
        }

        private function parcoursFichiers ( pEvt:MouseEvent ):void
        {
            // ouvre la boîte de dialogue permettant de parcourir les
fichiers
            envoyeurMultiple.browse ();
        }
    }
}

```

Lors du transfert des fichiers, la jauge de transfert indique l'état du chargement. En revanche, lors du transfert de multiples fichiers la jauge reçoit les événements de chaque fichier en cours de transfert et indique le chargement de tous les fichiers en même temps.

De la même manière, nous ne pouvons pas savoir simplement, si le transfert de tous les fichiers est terminé. Nous ne bénéficions pas d'événement approprié. Pour remédier à tout cela nous allons développer notre propre version de la classe `FileReferenceList`.

## A retenir

- Seule la classe `FileReferenceList` permet l'envoi de plusieurs fichiers en même temps.

## Création de la classe `EnvoiMultiple`

Nous allons améliorer la gestion d'envoi des fichiers en créant une classe `EnvoiMultiple`.

Celle-ci étend la classe `FileReferenceList` et améliore le transfert de chaque fichier en s'assurant que chaque fichier est transféré l'un après l'autre. De plus, nous allons ajouter un événement indiquant la fin du transfert de tous les fichiers.

Nous commençons par définir notre classe `EnvoiMultiple` en étendant la classe `FileReferenceList` :

```
package org.bytearray.envoi

{

    import flash.net.FileReferenceList;
    import flash.events.Event;
    import flash.net.URLRequest;

    public class EnvoiMultiple extends FileReferenceList

    {

        private var requete:URLRequest;

        public function EnvoiMultiple ( pRequete:URLRequest )

        {

            requete = pRequete;

            addEventListener ( Event.SELECT, selectionFichiers );

        }

        private function selectionFichiers ( pEvt:Event ):void

        {

            trace("fichiers sélectionnés");

        }

    }

}
```

La classe `EnvoiMultiple` accepte en paramètre un objet `URLRequest` pointant vers un script serveur permettant l'enregistrement du fichier.

Nous souhaitons que la classe `EnvoiMultiple` puisse gérer de manière synchronisée chaque envoi de fichiers. Pour cela nous allons lancer le premier transfert puis attendre que celui soit terminé pour déclencher les suivants :

```
package org.bytearray.envoi

{

    import flash.events.Event;
    import flash.events.ProgressEvent;
    import org.bytearray.events.ProgressQueueEvent;
    import org.bytearray.events.QueueEvent;
    import flash.net.FileReference;
    import flash.net.URLRequest;

    public class EnvoiMultiple extends FileReferenceList

    {

        private var historiqueFichiers:Array;
        private var fichierEnCours:FileReference;
        private var fichiers:Array;
        private var requete:URLRequest;

        public function EnvoiMultiple ( pRequete:URLRequest )

        {

            requete = pRequete;

            addEventListener ( Event.SELECT, selectionFichiers );

        }

        private function selectionFichiers ( pEvt:Event ):void

        {

            historiqueFichiers = new Array();

            fichiers = pEvt.target.fileList;

            suivant();

        }

        private function suivant ( ):void

        {

            var fichierEnCours:FileReference = fichiers.shift();

            historiqueFichiers.push ( fichierEnCours );

            fichierEnCours.addEventListener ( Event.COMPLETE,
transfertTermine );
            fichierEnCours.addEventListener ( ProgressEvent.PROGRESS,
transfertEnCours );

            fichierEnCours.upload ( requete );
```

```

    }

    private function transfertTermine ( pEvt:Event ):void
    {

    }

    private function transfertEnCours ( pEvt:ProgressEvent ):void
    {

    }

}

```

Lorsque le premier fichier a été transféré la méthode écouteur `transfertTermine` est déclenchée. Nous devons ajouter au sein de celle-ci le code nécessaire afin de lancer le transfert suivant, si cela est nécessaire. Nous en profitons pour supprimer l'écoute des événements `Event.COMPLETE` et `ProgressEvent.PROGRESS` auprès de l'objet `FileReference` en cours.

Nous consommons le tableau `fichierEnCours` en déclenchant la méthode `suivant`, si d'autres fichiers sont en attente de transfert au sein du tableau `fichiers` :

```

private function transfertTermine ( pEvt:Event ):void
{
    pEvt.target.removeEventListener ( Event.COMPLETE, transfertTermine );
    pEvt.target.removeEventListener ( ProgressEvent.PROGRESS,
    transfertEnCours );

    if ( fichiers.length ) suivant();

    trace("transfert terminé !");
}

private function transfertEnCours ( pEvt:ProgressEvent ):void
{
    trace("transfert en cours...");
}

```

Il ne nous reste plus qu'à diffuser deux événements pour chaque phase :

- `EvenementEnvoiMultiple.TERMINE` : cet événement est diffusé lorsque la totalité des fichiers sélectionnés sont transférés.

- `ProgressionEvenementEnvoiMultiple.PROGRESSION` : cet événement est diffusé lorsqu'un fichier est en cours de transfert.

Au sein du paquetage `evenements` nous créons la classe `EvenementEnvoiMultiple` :

```
package org.bytearray.evenements

{

    import flash.events.Event;

    public class EvenementEnvoiMultiple extends Event

    {

        public static const TERMINE:String = "termine";
        public var historique:Array;

        public function EvenementEnvoiMultiple ( pType:String,
        pHistorique:Array=null )

        {

            // initialisation du constructeur de la classe Event
            super( pType, false, false );

            historique = pHistorique;

        }

        // la méthode clone doit être surchargée
        public override function clone ():Event

        {

            return new EvenementEnvoiMultiple ( type, historique );

        }

        // la méthode toString doit être surchargée
        public override function toString ():String

        {

            return '[EvenementEnvoiMultiple type="'+ type +'\" bubbles=' +
            bubbles + ' eventPhase='+ eventPhase + ' cancelable=' + cancelable +']';

        }

    }

}
```

Puis la classe `ProgressionEvenementEnvoiMultiple` :

```
package org.bytearray.evenements

{

    import flash.events.ProgressEvent;
```

```

import flash.events.Event;
import flash.net.FileReference;

public class ProgressionEvenementEnvoiMultiple extends ProgressEvent
{
    public static const PROGRESSION:String = "progression";
    public var fichier:FileReference;

    public function ProgressionEvenementEnvoiMultiple ( pType:String,
pOctetsCharges:Number, pOctetsTotaux:Number, pFichier:FileReference )
    {
        // initialisation du constructeur de la classe Event
        super( pType, false, false, pOctetsCharges, pOctetsTotaux );

        fichier = pFichier;
    }

    // la méthode clone doit être surchargée
    public override function clone ():Event
    {
        return new ProgressionEvenementEnvoiMultiple ( type, bytesLoaded,
bytesTotal, fichier );
    }

    // la méthode toString doit être surchargée
    public override function toString ():String
    {
        return '[ProgressionEvenementEnvoiMultiple type="'+ type +'"
bubbles=' + bubbles + ' eventPhase='+ eventPhase + ' cancelable=' +
cancelable + ' bytesLoaded=' + bytesLoaded + ' bytesTotal=' + bytesTotal +']';
    }
}

```

Enfin nous diffusons les événements appropriés au sein des méthodes **transfertEnCours** et **transfertTermine** :

```

package org.bytearray.envoi
{
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import org.bytearray.evenements.EvenementEnvoiMultiple;
    import org.bytearray.evenements.ProgressionEvenementEnvoiMultiple;
    import flash.net.FileReference;
    import flash.net.FileReferenceList;
    import flash.net.URLRequest;

    public class EnvoiMultiple extends FileReferenceList

```



```

{

    private var historiqueFichiers:Array;
    private var fichierEnCours:FileReference;
    private var fichiers:Array;
    private var requete:URLRequest;

    public function EnvoiMultiple ( pRequete:URLRequest )

    {

        requete = pRequete;

        addEventListener ( Event.SELECT, selectionFichiers );

    }

    private function selectionFichiers ( pEvt:Event ):void

    {

        historiqueFichiers = new Array();

        fichiers = pEvt.target.fileList;

        suivant();

    }

    private function suivant ( ):void

    {

        var fichierEnCours:FileReference = fichiers.shift();

        historiqueFichiers.push ( fichierEnCours );

        fichierEnCours.addEventListener ( Event.COMPLETE,
transfertTermine );
        fichierEnCours.addEventListener ( ProgressEvent.PROGRESS,
transfertEnCours );

        fichierEnCours.upload ( requete );

    }

    private function transfertTermine ( pEvt:Event ):void

    {

        pEvt.target.removeEventListener ( Event.COMPLETE,
transfertTermine );
        pEvt.target.removeEventListener ( ProgressEvent.PROGRESS,
transfertEnCours );

        if ( fichiers.length ) suivant();

        else dispatchEvent ( new EvenementEnvoiMultiple (
EvenementEnvoiMultiple.TERMINER, historiqueFichiers ) );

    }

}

```

```

        private function transfertEnCours ( pEvt:ProgressEvent ):void
        {

            dispatchEvent ( new ProgressionEvenementEnvoiMultiple (
ProgressionEvenementEnvoiMultiple.PROGRESSION, pEvt.bytesLoaded,
pEvt.bytesTotal, FileReference ( pEvt.target ) ) );

        }
    }
}

```

Nous pouvons à présent utiliser la classe `EnvoiMultiple`. Dans un nouveau document Flash CS3, nous associons la classe de document suivante :

```

package org.bytearray.document
{

    import org.bytearray.abstrait.ApplicationDefaut;
    import org.bytearray.envoi.EnvoiMultiple;
    import org.bytearray.evenements.EvenementEnvoiMultiple;
    import org.bytearray.evenements.ProgressionEvenementEnvoiMultiple;
    import flash.display.MovieClip;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.text.TextField;

    public class Document extends ApplicationDefaut
    {

        private var envoyeur:EnvoiMultiple;
        public var boutonEnvoyer:SimpleButton;
        public var prechargeur:MovieClip;
        public var legende:TextField;

        public function Document ()
        {

            // création de l'objet EnvoiMultiple
            envoyeur = new EnvoiMultiple( new URLRequest
("http://www.bytearray.org/as3/upload.php" ));

            // écoute des événements personnalisés liés au chargement
            envoyeur.addEventListener (
ProgressionEvenementEnvoiMultiple.PROGRESSION, transfertEnCours );
            envoyeur.addEventListener ( EvenementEnvoiMultiple.TERMEINE,
transfertTermine );

            boutonEnvoyer.addEventListener ( MouseEvent.CLICK,
parcoursFichiers );

        }

        private function transfertEnCours (
pEvt:ProgressionEvenementEnvoiMultiple ):void

```

```

    {
        // mise à jour de la barre de progression
        prechargeur.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;

        // informations liées au fichier en cours de transfert
        legende.text = "Transfert en cours : " + pEvt.fichier.name;
    }

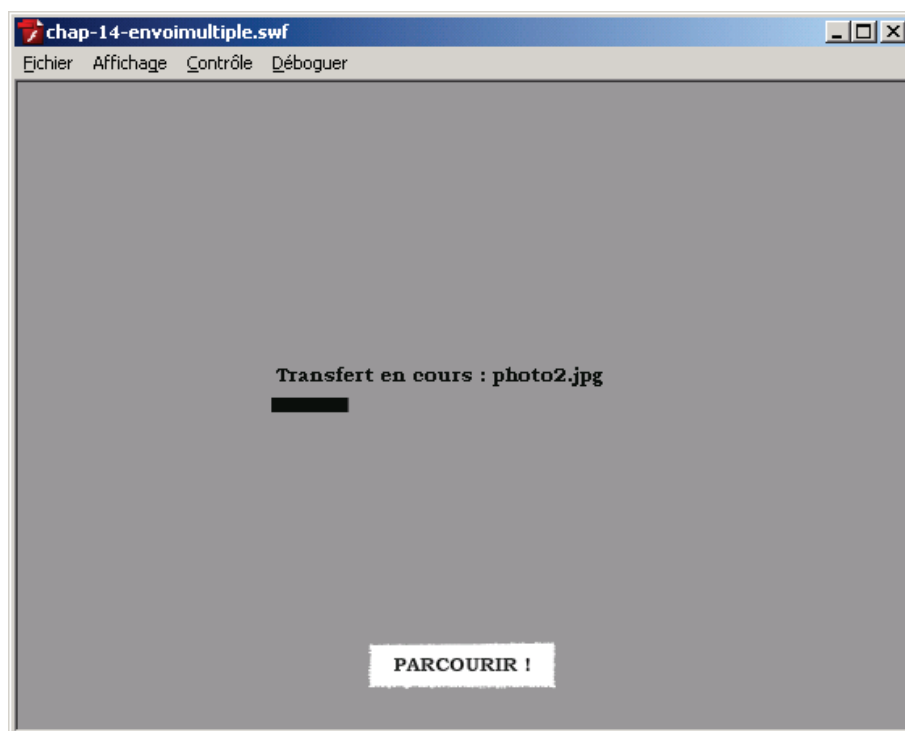
    private function transfertTermine ( pEvt:EvenementEnvoiMultiple ):void
    {
        // indique le nombre de fichiers transférés
        legende.text = pEvt.historique.length + " fichiers(s)
transféré(s)";
    }

    private function parcoursFichiers ( pEvt:MouseEvent ):void
    {
        envoyeur.browse();
    }
}
}

```

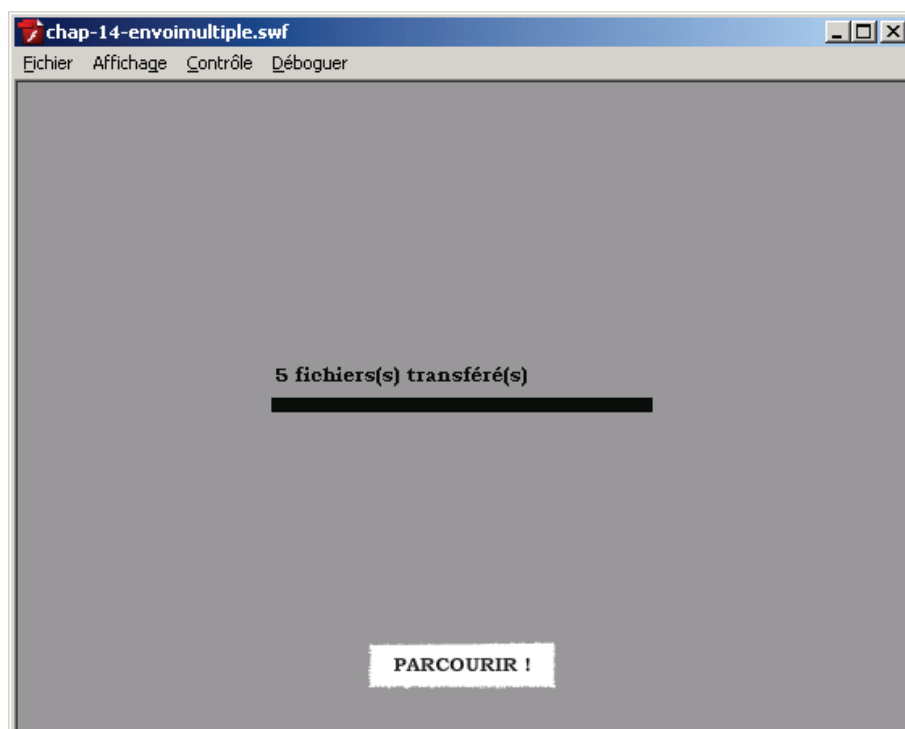
La document en cours contient une barre de préchargement `prechargeur` ainsi qu'un champ texte `legende` posés sur la scène afin d'afficher les informations relatives au transfert des fichiers.

La figure 14-19 illustre l'application :



*Figure 14-19. Transfert de fichiers.*

Lorsque les images ont été transférées, l'événement `EvenementEnvoiMultiple.TERMINE` est diffusé, cela nous permet d'afficher un message approprié et de réagir en conséquence :



*Figure 14-20. Transfert de fichiers terminé.*

La propriété `historique` de l'objet événementiel `EvenementEnvoiMultiple` est un tableau contenant des objets `FileReference` associé à chaque fichier transféré. En récupérant sa longueur nous pouvons indiquer le nombre de fichiers transférés.

A vous d'ajouter à présent, une instanciation dynamique de la jauge de transfert et de la légende puis de les supprimer une fois l'événement `EvenementEnvoiMultiple.TERMINÉ` est diffusé.

## Retourner des données une fois l'envoi terminé

ActionScript 3 ajoute un événement très intéressant à la classe `FileReference` que nous allons découvrir à présent.

Dans les précédentes versions d'ActionScript, il était impossible d'obtenir des informations du serveur lorsque le transfert était terminé. Très souvent, les développeurs souhaitaient retourner des informations indiquant si le transfert des données avait échoué ou non.

L'événement `FileReference.COMPLETE` n'offre pas la possibilité de renvoyer les informations provenant du serveur. En revanche, ActionScript 3 introduit un événement `DataEvent.UPLOAD_COMPLETE_DATA` offrant cette possibilité.

Nous allons ajouter cette fonctionnalité à notre classe `EnvoiMultiple` et renvoyer des informations afin d'indiquer à l'application Flash que l'écriture sur le serveur a réussi ou non.

Nous allons modifier le script serveur gérant l'enregistrement des fichiers de manière à indiquer si l' :

```
<?php

$fichier = $_FILES["Filedata"];

if ( isset ( $fichier ) )

{

    $nomFichier = $fichier['name'];

    if ( move_uploaded_file ( $fichier['tmp_name'], $nomFichier) ) echo
"resultat=1";

    else echo "resultat=0";

} else echo "resultat=0";

?>
```

Voici le code complet de la classe `EnvoiMultiple` :

```

package org.bytearray.envoi
{
    import flash.events.DataEvent;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import org.bytearray.evenements.EvenementEnvoiMultiple;
    import org.bytearray.evenements.ProgressionEvenementEnvoiMultiple;
    import flash.net.FileReference;
    import flash.net.FileReferenceList;
    import flash.net.URLRequest;

    public class EnvoiMultiple extends FileReferenceList
    {
        private var historiqueFichiers:Array;
        private var fichierEnCours:FileReference;
        private var fichiers:Array;
        private var requete:URLRequest;

        public function EnvoiMultiple ( pRequete:URLRequest )
        {
            requete = pRequete;

            addEventListener ( Event.SELECT, selectionFichiers );
        }

        private function selectionFichiers ( pEvt:Event ):void
        {
            historiqueFichiers = new Array();

            fichiers = pEvt.target.fileList;

            suivant();
        }

        private function suivant ( ):void
        {
            var fichierEnCours:FileReference = fichiers.shift();

            historiqueFichiers.push ( fichierEnCours );

            fichierEnCours.addEventListener ( Event.COMPLETE,
transfertTermine );
            fichierEnCours.addEventListener ( ProgressEvent.PROGRESS,
transfertEnCours );
            fichierEnCours.addEventListener ( DataEvent.UPLOAD_COMPLETE_DATA,
retourServeur );

            fichierEnCours.upload ( requete );
        }
    }
}

```

```

        private function retourServeur ( pEvt:DataEvent ):void
        {
            dispatchEvent ( pEvt );

            pEvt.target.removeEventListener ( DataEvent.UPLOAD_COMPLETE_DATA,
retourServeur );
        }

        private function transfertTermine ( pEvt:Event ):void
        {
            pEvt.target.removeEventListener ( Event.COMPLETE,
transfertTermine );
            pEvt.target.removeEventListener ( ProgressEvent.PROGRESS,
transfertEnCours );

            if ( fichiers.length ) suivant();

            else dispatchEvent ( new EvenementEnvoiMultiple (
EvenementEnvoiMultiple.TERME, historiqueFichiers ) );
        }

        private function transfertEnCours ( pEvt:ProgressEvent ):void
        {
            dispatchEvent ( new ProgressionEvenementEnvoiMultiple (
ProgressionEvenementEnvoiMultiple.PROGRESSION, pEvt.bytesLoaded,
pEvt.bytesTotal, FileReference ( pEvt.target ) ) );
        }
    }
}

```

Il ne reste plus qu'à écouter cet événement auprès de l'objet **EnvoiMultiple** créé :

```

package org.bytearray.document
{
    import flash.events.DataEvent;
    import flash.net.URLRequest;
    import flash.net.URLVariables;
    import org.bytearray.abstrait.ApplicationDefaut;
    import org.bytearray.envoi.EnvoiMultiple;
    import org.bytearray.evenements.EvenementEnvoiMultiple;
    import org.bytearray.evenements.ProgressionEvenementEnvoiMultiple;
    import flash.display.MovieClip;
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.text.TextField;

    public class Document extends ApplicationDefaut
    {
        private var _fichiers:Array<FileReference> = [];
        private var _historiqueFichiers:Array<FileReference> = [];
        private var _suivant:Function = null;
        private var _transfertTermine:Function = null;
        private var _transfertEnCours:Function = null;
        private var _progression:ProgressionEvenementEnvoiMultiple = null;
        private var _evenementEnvoiMultiple:EvenementEnvoiMultiple = null;
        private var _movieClip:MovieClip = null;
        private var _simpleButton:Simp

```

```

{

    private var envoyeur:EnvoiMultiple;
    public var boutonEnvoyer:SimpleButton;
    public var prechargeur:MovieClip;
    public var legende:TextField;

    public function Document ()

    {

        // création de l'objet EnvoiMultiple
        envoyeur = new EnvoiMultiple( new URLRequest
("http://www.bytearray.org/as3/upload.php" ));

        // écoute des événements personnalisés liés au chargement
        envoyeur.addEventListener (
ProgressionEvenementEnvoiMultiple.PROGRESSION, transfertEnCours );
        envoyeur.addEventListener ( EvenementEnvoiMultiple.TERMEINE,
transfertTermine );
        envoyeur.addEventListener ( DataEvent.UPLOAD_COMPLETE_DATA,
retourServeur );

        boutonEnvoyer.addEventListener ( MouseEvent.CLICK,
parcoursFichiers );

    }

    private function retourServeur ( pEvt:DataEvent ):void

    {

        var variables:URLVariables = new URLVariables ( pEvt.data );

        var retour:Number = Number ( variables.resultat );

        legende.text = retour ? "Transfert réussi !" : "Echec de
transfert";

    }

    private function transfertEnCours (
pEvt:ProgressionEvenementEnvoiMultiple ):void

    {

        // mise à jour de la barre de progression
        prechargeur.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;

        // informations liées au fichier en cours de transfert
        legende.text = "Transfert en cours : " + pEvt.fichier.name;

    }

    private function transfertTermine ( pEvt:EvenementEnvoiMultiple ):void

    {

        // indique le nombre de fichiers transférés
        legende.text = pEvt.historique.length + " fichiers(s)
transféré(s)";

    }

}

```



```
    }  
  
    private function parcoursFichiers ( pEvt:MouseEvent ):void  
    {  
  
        envoyeur.browse();  
  
    }  
}  
}
```

Vous remarquerez que nous utilisons la classe `URLVariables` afin de décoder la chaîne encodée URL retournée par le serveur.

Nous avons achevé notre classe `EnvoiMultiple` qui pourra être réutilisée dans tous les projets nécessitant un transfert de fichiers synchronisés.

## A retenir

- La classe `FileReferenceList` envoie tous les fichiers en même temps et ne diffuse pas d'événement approprié lorsque tous les fichiers ont été transférés.
- Afin de corriger cela, nous avons créé une classe `EnvoiMultiple`.

## Aller plus loin

Souvenez-vous, lors du chapitre 10 intitulé *Diffusion d'événements personnalisés* nous avons entamé la conception d'une classe `ChargeurXML` pouvant charger un flux XML et diffuser un événement `Event.COMPLETE` :

```
package  
  
{  
  
    import flash.events.Event;  
    import flash.events.EventDispatcher;  
    import flash.net.URLRequest;  
  
    public class ChargeurXML extends EventDispatcher  
    {  
  
        public function ChargeurXML ()  
        {  
  
        }  
  
        public function charge ( pRequete:URLRequest ):void
```

```

        {
        }

        public function chargementTermine ( pEvt:Event ):void

        {
        }

    }
}

```

Nous allons la compléter en ajoutant créant un objet **URLoader** interne :

```

package org.bytearray.xml

{

    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.HTTPStatusEvent;
    import flash.events.SecurityErrorEvent;
    import flash.events.EventDispatcher;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;

    public class ChargeurXML extends EventDispatcher

    {

        private var chargeur:URLLoader;
        private var fluxXML:XML;

        public function ChargeurXML ()

        {

            chargeur = new URLLoader();

            chargeur.dataFormat = URLLoaderDataFormat.TEXT;

            chargeur.addEventListener ( Event.OPEN, redirigeEvenement );
            chargeur.addEventListener ( ProgressEvent.PROGRESS,
redirigeEvenement );
            chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
            chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS,
redirigeEvenement );
            chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );
            chargeur.addEventListener ( SecurityErrorEvent.SECURITY_ERROR,
redirigeEvenement );

        }

        private function redirigeEvenement ( pEvt:Event ):void

        {

```

```

        dispatchEvent ( pEvt );
    }

    public function charge ( pRequete:URLRequest ):void
    {
        chargeur.load ( pRequete );
    }

    private function chargementTermine ( pEvt:Event ):void
    {
        try
        {
            fluxXML = new XML ( pEvt.target.data );
        } catch ( pErreur:Error )
        {
            trace (pErreur);
        }

        dispatchEvent ( pEvt );
    }

    public function get donnees ( ):XML
    {
        return fluxXML;
    }
}

```

Une fois le fichier XML chargé nous diffusons un événement `Event.COMPLETE`. Ainsi, le code permettant de charger un fichier XML est extrêmement réduit :

```

package org.bytearray.document
{
    import org.bytearray.abstrait.ApplicationDefaut;
    import org.bytearray.xml.ChargeurXML;
    import flash.events.Event;
    import flash.net.URLRequest;

    public class Document extends ApplicationDefaut
    {

```

```

        public function Document ()
        {
            var chargeur:ChargeurXML = new ChargeurXML ();

            chargeur.charge ( new URLRequest ("donnees.xml") );

            chargeur.addEventListener ( Event.COMPLETE, chargementTermine );

        }

        private function chargementTermine ( pEvt:Event ):void
        {
            /* affiche :
            <MENU>
            <BOUTON legende="Accueil" couleur="0x887400" vitesse="1"
url="accueil.swf"/>
            <BOUTON legende="Photos" couleur="0x005587" vitesse="1"
url="photos.swf"/>
            <BOUTON legende="Blog" couleur="0x125874" vitesse="1"
url="blog.swf"/>
            <BOUTON legende="Liens" couleur="0x59CCAA" vitesse="1"
url="liens.swf"/>
            <BOUTON legende="Forum" couleur="0xEE44AA" vitesse="1"
url="forum.swf"/>
            </MENU>
            */
            trace( pEvt.target.donnees );

        }

    }
}

```

La classe `ChargeurXML` peut ainsi être utilisée dans de nombreux projets ayant recourt au XML. Si un matin, un membre de votre équipe se plaint de la mise en cache des XML par le navigateur.

Assurez-lui, qu'un système anti-cache peut être ajouté à la classe `ChargeurXML` :

```

package org.bytearray.xml
{
    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.HTTPStatusEvent;
    import flash.events.ProgressEvent;
    import flash.events.SecurityErrorEvent;
    import flash.events.EventDispatcher;
    import flash.net.URLLoader;
    import flash.net.URLLoaderDataFormat;
    import flash.net.URLRequest;
    import flash.net.URLRequestHeader;
}

```

```

public class ChargeurXML extends EventDispatcher
{
    private var chargeur:URLLoader;
    private var fluxXML:XML;
    private var antiCache:Boolean;

    public function ChargeurXML ( pAntiCache:Boolean=false )
    {
        antiCache = pAntiCache;

        chargeur = new URLLoader();

        chargeur.dataFormat = URLLoaderDataFormat.TEXT;

        chargeur.addEventListener ( Event.OPEN, redirigeEvenement );
        chargeur.addEventListener ( ProgressEvent.PROGRESS,
redirigeEvenement );
        chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
        chargeur.addEventListener ( HTTPStatusEvent.HTTP_STATUS,
redirigeEvenement );
        chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );
        chargeur.addEventListener ( SecurityErrorEvent.SECURITY_ERROR,
redirigeEvenement );
    }

    public function charge ( pRequete:URLRequest ):void
    {
        if ( antiCache ) pRequete.requestHeaders.push ( new
URLRequestHeader ( "pragma", "no-cache" ) );

        chargeur.load ( pRequete );
    }

    private function redirigeEvenement ( pEvt:Event ):void
    {
        dispatchEvent ( pEvt );
    }

    private function chargementTermine ( pEvt:Event ):void
    {
        try
        {
            fluxXML = new XML ( pEvt.target.data );
        } catch ( pErreur:Error )

```

```
        {  
            trace (pErreur);  
        }  
        dispatchEvent ( pEvt );  
    }  
    public function get donnees ( ):XML  
    {  
        return fluxXML;  
    }  
}
```

Grace à la classe `URLRequestHeader`, nous avons modifié l'entête HTTP du lecteur Flash lors du chargement du fichier XML et empêché sa mise en cache par le navigateur.

Ainsi chaque développeur souhaitant tirer profit de cette fonctionnalité devra simplement le préciser lors de l'instanciation de l'objet `ChargeurXML` :

```
// crée un objet ChargeurXML en précisant de jamais mettre en cache les  
fichiers XML chargés  
var chargeur:ChargeurXML = new ChargeurXML ( true );
```

D'autres entêtes HTTP peuvent être utilisées, nous reviendrons sur la classe `URLRequestHeader` au cours du chapitre 20 intitulé `ByteArray`.

## A retenir

- La classe `URLRequestHeader` permet de modifier les entêtes HTTP du lecteur Flash.

Dans le prochain chapitre, nous explorerons les nouveautés apportées par ActionScript 3 en matière d'échanges entre le lecteur Flash et les différents navigateurs.

# 15

## Communication Externe

<b>CONTENEUR ET CONTENU .....</b>	<b>1</b>
<b>PASSER DES VARIABLES .....</b>	<b>2</b>
INTÉGRATION PAR JAVASCRIPT .....	3
LA PROPRIÉTÉ PARAMETERS .....	5
LES FLASHVARS .....	10
PASSER DES VARIABLES DYNAMIQUES .....	11
ACCÉDER FACILEMENT AUX FLASHVARS .....	14
<b>APPELER UNE FONCTION .....</b>	<b>17</b>
<b>L'API EXTERNALINTERFACE .....</b>	<b>18</b>
APPELER UNE FONCTION EXTERNE DEPUIS ACTIONSCRIPT .....	22
APPELER UNE FONCTION ACTIONSCRIPT DEPUIS LE CONTENEUR .....	27
COMMUNICATION ET SÉCURITÉ .....	30

### Conteneur et contenu

Lors du déploiement d'une application Flash sur Internet, nous intégrons généralement celle-ci au sein d'une page conteneur HTML interprétée par différents navigateurs tels Internet Explorer, Firefox, Opera ou autres. Dans d'autres situations, celle-ci peut être intégrée au sein d'une application bureau développée en C#, C++ ou Java.

Bien qu'autonome, l'animation lue par le lecteur Flash peut nécessiter des informations provenant de l'application conteneur dans le cas de développement d'applications dynamiques.

Avant de s'intéresser à la notion d'échanges, il convient de définir dans un premier temps les deux acteurs concernés par une éventuelle communication :

- Le conteneur : Une application contenant le SWF. Il peut s'agir d'une page HTML ou d'une application C#, C++ ou autres.
- L'application : Il s'agit du SWF lu par le lecteur Flash.

Au cours de ce chapitre, nous allons nous intéresser aux différentes techniques de communication possible entre ces deux acteurs tout en se préoccupant des éventuelles restrictions de sécurité pouvant intervenir.

## Passer des variables

Afin d'introduire la notion d'échanges entre ces deux acteurs, nous allons nous intéresser dans un premier temps au passage de simples variables encodées au format URL.

Afin de bien comprendre l'intérêt de tels échanges, mettons nous en situation avec le cas suivant :

Vous devez développer un lecteur vidéo qui puisse charger différentes vidéos dynamiques pour un portail destiné aux cinéphiles. Afin de rendre le lecteur le plus dynamique possible, il serait judicieux de pouvoir spécifier en dehors de l'animation quelle bande-annonce jouer.

Vous pensez dans un premier temps à la création d'un fichier XML contenant le nom de la vidéo à jouer. Certes, cette technique fonctionnerait mais n'est pas optimisée. Le fichier XML devrait être dupliqué pour chaque lecteur vidéo, rendant le développement rigide et redondant.

La solution la plus efficace consiste à passer dynamiquement le nom de la vidéo à jouer depuis la page navigateur contenant le lecteur vidéo. Pour cela, deux techniques existent :

La première consiste à passer des variables encodées URL après le nom du fichier au sein du paramètre `movie` de la balise `object` et `src` de la balise `embed` :

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab
#version=9,0,0,0" width="550" height="400" id="chap-15-variables"
align="middle">
  <param name="allowScriptAccess" value="sameDomain" />
  <param name="allowFullScreen" value="false" />
  <param name="movie" value="chap-15-variables.swf?maVar1=15&maVar2=50"
/><param name="quality" value="high" /><param name="bgcolor" value="#ffffff"
/>
  <embed src="chap-15-variables.swf?maVar1=15&maVar2=50" quality="high"
bgcolor="#ffffff" width="550" height="400" name="chap-15-variables"
align="middle" allowScriptAccess="sameDomain" allowFullScreen="false"
```



```
type="application/x-shockwave-flash"
pluginspage="http://www.macromedia.com/go/getflashplayer" />
</object>
```

Automatiquement, les variables `maVar1` et `maVar2` sont accessibles au sein du SWF intitulé `chap-15-variables.swf`.

Cette technique est utilisée lorsque l'animation est intégrée manuellement au sein d'une page conteneur.

Dans le cas de Flash CS3 un script JavaScript est généré automatiquement lors de la publication afin d'intégrer dynamiquement l'animation. Flash CS3 intègre un mécanisme d'intégration de l'animation Flash par script JavaScript permettant de ne pas avoir à demander l'autorisation de l'utilisateur afin de lire du contenu Flash.

Cette astuce évite d'avoir à activer l'animation en cliquant dessus au sein d'Internet Explorer.

---

En avril 2007, la société Eolas avait obtenu de Microsoft la modification d'Internet Explorer visant à demander obligatoirement l'activation de l'utilisateur lorsqu'un contenu ActiveX était intégré dans une page.

---

Si la page contenant l'animation est directement générée depuis Flash CS3, la technique visant à intégrer les variables directement au sein des balises `object` et `embed` ne fonctionnera pas.

Il nous faut passer les variables depuis la fonction JavaScript `AC_FL_RunContent`.

## A retenir

- L'utilisation de variables encodées au format URL est le moyen le plus simple de passer des données au SWF.
- Lorsque la page conteneur est générée depuis Flash CS3, l'intégration du SWF dans la page est gérée par la fonction JavaScript `AC_FL_RunContent`.

## Intégration par JavaScript

Lorsque la page conteneur est directement générée depuis Flash CS3, le script suivant est intégré à la page HTML afin d'intégrer le SWF :

```
<script language="javascript">
  if (AC_FL_RunContent == 0) {
    alert("Cette page nécessite le fichier AC_RunActiveContent.js.");
  } else {
    AC_FL_RunContent(
      'codebase',
      'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=
      9,0,0,0',
```

```
        'width', '550',
        'height', '400',
        'src', 'chap-15-variables',
        'quality', 'high',
        'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
        'align', 'middle',
        'play', 'true',
        'loop', 'true',
        'scale', 'showall',
        'wmode', 'window',
        'devicefont', 'false',
        'id', 'chap-15-variables',
        'bgcolor', '#ffffff',
        'name', 'chap-15-variables',
        'menu', 'true',
        'allowFullScreen', 'false',
        'allowScriptAccess', 'sameDomain',
        'movie', 'chap-15-variables',
        'salign', ''
    );
}
</script>
```

La fonction `AC_FL_RunContent` intègre l’animation en ajoutant chaque attribut passé en paramètre.

Ainsi, pour passer les variables équivalentes à l’exemple précédent, nous passons les variables au sein du paramètre `movie` de la fonction `AC_FL_RunContent` :

```
AC_FL_RunContent(
    'codebase',
    'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=
9,0,0,0',
    'width', '550',
    'height', '400',
    'src', 'chap-15-external-interface',
    'quality', 'high',
    'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
    'align', 'middle',
    'play', 'true',
    'loop', 'true',
    'scale', 'showall',
    'wmode', 'window',
    'devicefont', 'false',
    'id', 'monApplication',
    'bgcolor', '#ffffff',
    'name', 'monApplication',
    'menu', 'true',
    'allowFullScreen', 'false',
    'allowScriptAccess', 'sameDomain',
    'movie', 'chap-15-variables?maVar1=15&maVar2=50',
    'salign', ''
);
```

Dans les précédentes versions d’ActionScript, les variables étaient directement placées sur le scénario principal `_root`.

En ActionScript 3, afin d'éviter les conflits de variables, celles-ci ne sont plus disponibles à partir du scénario principal mais depuis la propriété `parameters` de l'objet `LoaderInfo` du scénario du SWF.

## A retenir

- Dans le cas de l'utilisation de la fonction `AC_FL_RunContent`, les variables doivent être passées au sein du paramètre `movie`.

## La propriété `parameters`

Comme nous l'avons vu au cours du chapitre 13 intitulé *Chargement de contenu*. Chaque SWF contient un objet `LoaderInfo` associé, contenant différentes informations.

Dans le cas de variables passées par le navigateur, les variables passées dynamiquement deviennent des propriétés de l'objet référencé par la propriété `parameters` de l'objet `LoaderInfo`.

Dans le code suivant nous définissons la classe de document suivante, afin d'accéder aux deux variables passées :

```
package org.bytearray.document

{

    import flash.text.TextField;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault

    {

        public function Document ()

        {

            var infos:Object = root.loaderInfo.parameters;

            infosVariables.appendText ( infos.maVar1 + "\n" );
            infosVariables.appendText ( infos.maVar2 + "\n" );

        }

    }

}
```

Un champ texte `infosVariables` est posé sur le scénario principal afin d'afficher les variables réceptionnées.

Lorsque des variables sont passées au lecteur Flash, les variables sont copiées au sein de la propriété `parameters` de l'objet `LoaderInfo`. Vous remarquez que nous accédons aux variables comme des propriétés de l'objet `parameters`.

---

Il est important de noter que les variables sont copiées au sein de l'objet `LoaderInfo` avant l'exécution du code. Celles-ci sont donc accessibles instantanément.

---

Dans le code précédent nous ciblons de manière explicite la propriété `loaderInfo` du scénario principal, à l'aide de la propriété `root`.

Dans un contexte de classe du document, l'instance en cours fait référence au scénario principal, nous pouvons donc directement accéder à l'objet `LoaderInfo` de la manière suivante :

```
package org.bytearray.document
{
    import flash.text.TextField;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault
    {
        public function Document ()
        {
            var infos:Object = loaderInfo.parameters;

            infosVariables.appendText ( infos.maVar1 + "\n" );
            infosVariables.appendText ( infos.maVar2 + "\n" );
        }
    }
}
```

Au cas où nous ne connaissons pas le nom des variables passées, nous pouvons les énumérer à l'aide d'une boucle `for in` :

```
package org.bytearray.document
{
    import flash.text.TextField;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault
    {
        public function Document ()
        {
            var infos:Object = loaderInfo.parameters;

            for ( var p:String in infos )
```

```
        {  
            infosVariables.appendText ( p + "\n" );  
        }  
    }  
}  
}
```

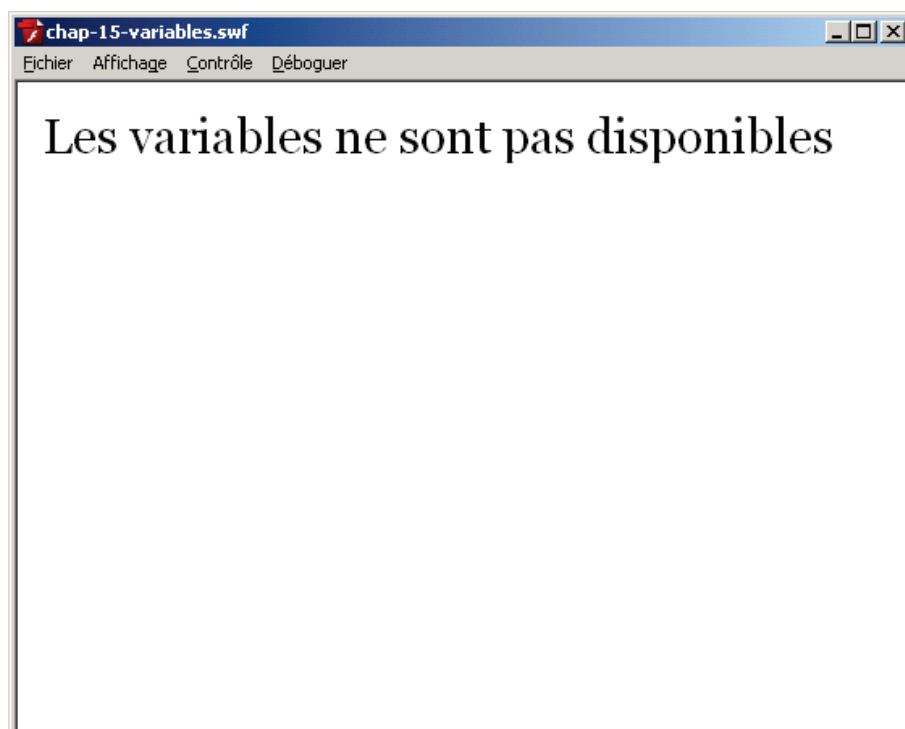
Si aucune variable n'est passée, la boucle n'est pas exécutée, aucun contenu n'est affiché dans le champ texte `infosVariables`.

Lors du test de l'animation au sein de Flash CS3, l'animation est lue au sein du lecteur autonome. Aucune variable ne peut donc être passée. De la même manière, si les variables sont passées d'une manière non appropriée, nous devons le gérer.

Il est donc important de s'assurer que les variables sont bien définies, pour cela nous écrivons le code suivant :

```
package org.bytearray.document  
{  
    import flash.text.TextField;  
    import org.bytearray.abstrait.ApplicationDefault;  
  
    public class Document extends ApplicationDefault  
    {  
        public function Document ()  
        {  
            var infos:Object = loaderInfo.parameters;  
  
            if ( infos.maVar1 != undefined && infos.maVar2 != undefined )  
            {  
                infosVariables.appendText ( infos.maVar1 + "\n" );  
                infosVariables.appendText ( infos.maVar2 + "\n" );  
            } else infosVariables.appendText ( "Les variables ne sont pas  
disponibles" );  
        }  
    }  
}
```

La figure 15-1 illustre le résultat :



*Figure 15-1. Variables non accessibles.*

Si nous testons l’animation au sein d’une page passant correctement les variables, nous obtenons le résultat illustré en figure 15-2 :



15  
50

*Figure 15-2. Variables correctement passées.*

Il est important de noter que les variables sont copiées en tant que chaîne de caractères. Ainsi, si nous souhaitons manipuler les variables afin de faire des opérations mathématiques nous devons au préalable le convertir en nombres.

Dans le code suivant, nous tentons d’additionner les deux variables, nous obtenons alors une simple concaténation :

```
| infosVariables.appendText ( infos.maVar1 + infos.maVar2 );
```

La figure 15-3 illustre le résultat :



1550

*Figure 15-3. Variables concaténées.*

Afin d'ajouter correctement les variables nous devons au préalable les convertir en tant que nombre :

```
infosVariables.appendText ( String ( Number ( infos.maVar1 ) + Number (
infos.maVar2 ) ) );
```

La figure 15-4 illustre le résultat :



65

*Figure 15-4. Variables additionnées.*

Cette technique offre en revanche une limitation de volume de données passées propre à chaque navigateur. Depuis le lecteur 6, nous avons la possibilité de passer ces mêmes variables à l'aide d'un nouvel attribut des balises `object` et `embed` appelé `flashvars`.

De plus, le passage de variables sans l'utilisation de l'attribut `flashvars` force le rechargement du SWF empêchant donc sa mise en cache. Notons que si les variables passées sont identiques à plusieurs reprises, il n'y a pas de rechargement forcé.

**A retenir**

- Les variables passées sont copiées au sein de l'objet retourné par la propriété `parameters` de l'objet `LoaderInfo` du scénario principal.
- Les variables sont copiées en tant que chaînes de caractères.
- Il convient de convertir les variables en nombre avant d'effectuer des opérations mathématiques sur chacune d'elles.

## Les FlashVars

L'utilisation des *FlashVars* est recommandée depuis Flash MX (Flash 6). Pour passer des variables grâce à l'attribut `flashvars`, il nous suffit d'ajouter l'attribut `flashvars` en paramètre de la fonction `AC_FL_RunContent` :

```
AC_FL_RunContent(
    'codebase',
    'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=
9,0,0,0',
    'width', '550',
    'height', '400',
    'flashvars', 'maVar1=15&maVar2=50',
    'src', 'chap-15-external-interface',
    'quality', 'high',
    'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
    'align', 'middle',
    'play', 'true',
    'loop', 'true',
    'scale', 'showall',
    'wmode', 'window',
    'devicefont', 'false',
    'id', 'monApplication',
    'bgcolor', '#ffffff',
    'name', 'monApplication',
    'menu', 'true',
    'allowFullScreen', 'false',
    'allowScriptAccess', 'sameDomain',
    'movie', 'chap-15-external-interface',
    'salign', ''
);
```

Si nous testons à nouveau l'animation les variables sont accessibles de la même manière.

## A retenir



- L'utilisation des *FlashVars* est recommandée depuis le lecteur Flash 6.
- Les balises `object` et `embed` possèdent un attribut `flashvars` permettant un passage de variables optimisé.

## Passer des variables dynamiques

Afin d'aller plus loin, nous allons récupérer dynamiquement les variables passées au sein de l'url de l'animation et les récupérer dans notre application.

Dans le cas d'un site Flash traditionnel, nous souhaitons pouvoir récupérer les variables passées en GET depuis l'url suivante :

```
http://www.monsite.org/index.php?rubrique=4&langue=fr
```

Le lecteur Flash n'a pas la capacité de les récupérer de manière autonome, nous pouvons utiliser pour cela un script JavaScript ou autre qui se chargera de les passer au SWF grâce à l'attribut `flashvars`.

Pour cela, nous ajoutons au sein de la page conteneur une fonction JavaScript nommée `recupVariables` :

```
<script language="javascript">

var position = window.location.href.indexOf ("?")+1;
var chaine = window.location.href.substr ( position );

if (AC_FL_RunContent == 0) {
    alert("Cette page nécessite le fichier AC_RunActiveContent.js.");
} else {
    AC_FL_RunContent(
        'codebase',
        'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,0,0',
        'width', '550',
        'height', '400',
        'flashvars', chaine,
        'src', 'chap-15-flashvars-dynamique',
        'quality', 'high',
        'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
        'align', 'middle',
        'play', 'true',
        'loop', 'true',
        'scale', 'showall',
        'wmode', 'window',
        'devicefont', 'false',
        'id', 'chap-15-flashvars-dynamique',
        'bgcolor', '#ffffff',
        'name', 'chap-15-flashvars-dynamique',
        'menu', 'true',
        'allowFullScreen', 'false',
        'allowScriptAccess', 'sameDomain',
        'movie', 'chap-15-flashvars-dynamique',
        'salign', ''
    ); //end AC code
```

```
}  
</script>
```

Nous passons dynamiquement les `flashvars` en reconstituant une chaîne encodée URL à l'aide des variables récupérées depuis l'URL.

Afin de tester si les variables sont bien passées, nous accédons à notre animation en ajoutant les variables encodées URL en fin d'adresse :

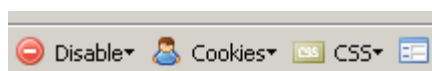
```
| http://www.monsite.org/index.php?rubrique=4&langue=fr
```

Afin d'éviter les erreurs nous testons si les variables sont définies :

```
package org.bytearray.document  
{  
    import flash.text.TextField;  
    import flash.display.MovieClip;  
  
    public class Document extends MovieClip  
    {  
        public function Document ()  
        {  
            var infos:Object = loaderInfo.parameters;  
  
            if ( (infos.rubrique != undefined && infos.langue != undefined) )  
            {  
                infosVariables.appendText ( "langue = " + infos.langue +  
                "\n" );  
                infosVariables.appendText ( "rubrique = " + infos.rubrique +  
                "\n" );  
            } else infosVariables.appendText ( "Les variables ne sont pas  
disponibles" );  
        }  
    }  
}
```

En testant notre animation, nous voyons que les variables sont correctement passées.

La figure 15-4 illustre le résultat :



langue = fr  
rubrique = 4

*Figure 15-5. FlashVars dynamiques.*

Souvenez-vous, les variables sont passées en tant que chaîne de caractères.

Ainsi, si aucune variable n'est passée dans l'url, la valeur des variables `rubrique` et `langue` seront bien différentes de `undefined`, car elles auront pour valeur `"undefined"` en tant que chaîne.

Pour gérer cela, nous ajoutons la condition suivante :

```
package org.bytearray.document

{

    import flash.text.TextField;
    import oflash.display.MovieClip;

    public class Document extends MovieClip

    {

        public function Document ()

        {

            var infos:Object = loaderInfo.parameters;

            if ( (infos.rubrique != undefined && infos.langue != undefined)
&&
                (infos.rubrique != "undefined" && infos.langue != "undefined") )

            {

                infosVariables.appendText ( "langue = " + infos.langue +
"\n" );
                infosVariables.appendText ( "rubrique = " + infos.rubrique +
"\n" );

            } else infosVariables.appendText ( "Les variables ne sont pas
disponibles" );

        }

    }

}
```

```
| }  
| }
```

Ainsi, si nous accédons à l'animation en omettant les variables au sein de l'URL, nous obtenons le message illustré en figure 15-6 :



Les variables ne sont pas disponibles

*Figure 15-6. Message d'erreur.*

Dans un contexte réel, il est important de s'assurer que chaque SWF composant le site puisse accéder sans problèmes aux variables passées au SWF principal.

Dans la partie suivante, nous allons découvrir comment faciliter le passage de celles-ci au reste de l'application.

## A retenir

- A l'aide d'un script intégré à la page conteneur, nous pouvons passer des variables dynamiques à l'animation.

### Accéder facilement aux FlashVars

Nous savons que les variables passées à l'animation sont accessibles depuis l'objet `LoaderInfo` du scénario principal.

Afin de garantir un accès simplifié de ces variables aux différents SWF composant notre application, nous allons diffuser un événement personnalisé par l'intermédiaire de la propriété `sharedEvents` de l'objet `LoaderInfo`.

Souvenez, vous au cours du chapitre 13 intitulé *Chargement de contenu*, nous avons vu que la propriété `sharedEvents` était une excellent passerelle de communication entre le SWF chargeant et chargé.

Nous définissons dans un premier temps une classe `InfosEvenement` chargée de transporter les variables :

```
package org.bytearray.evenements  
{  
    import flash.display.LoaderInfo;  
    import flash.events.Event;
```

```
public class InfosEvenement extends Event
{
    public static const INFOS:String = "infos";

    public var infos:LoaderInfo;

    public function InfosEvenement ( pType:String, pLoaderInfo:LoaderInfo
)
    {
        super ( pType, false, false );

        infos = pLoaderInfo;
    }

    public override function clone ( ):Event
    {
        return new InfosEvenement ( type, infos );
    }
}
}
```

Puis nous modifions la classe de document du SWF principal afin de charger dynamiquement le SWF :

```
package org.bytearray.document
{
    import flash.events.Event;
    import flash.net.URLRequest;
    import flash.display.Loader;
    import flash.display.MovieClip;
    import org.bytearray.evenements.InfosEvenement;

    public class Document extends MovieClip
    {
        private var loader:Loader;

        public function Document ()
        {
            loader = new Loader();

            loader.contentLoaderInfo.addEventListener ( Event.COMPLETE,
onComplete );

            loader.load ( new URLRequest ("chap-15-animation.swf") );
        }
    }
}
```

```
        addChild ( loader );

    }

    private function onComplete ( pEvt:Event ):void
    {

        pEvt.target.sharedEvents.dispatchEvent ( new InfosEvenement (
InfosEvenement.INFOFOS, loaderInfo ) );

    }

}

}
```

Afin de récupérer les variables envoyées, nous associons la classe de document suivante au SWF chargé :

```
package org bytearray rubriques
{

    import flash.events.Event;
    import flash.text.TextField;
    import flash.display.MovieClip;
    import org bytearray abstrait.ApplicationDefault;

    public class Rubrique extends ApplicationDefault
    {

        public var infosVariables:TextField;

        public function Rubrique ()
        {

            loaderInfo.sharedEvents.addEventListener ( InfosEvenement.INFOFOS,
infos );

        }

        function infos ( pEvt:InfosEvenement ):void
        {

            var infos:Object = pEvt.infos.parameters;

            if ( (infos.rubrique != undefined && infos.langue != undefined)
&&
            (infos.rubrique != "undefined" && infos.langue != "undefined") )
            {

                infosVariables.appendText ( infos.rubrique + "\n" );
                infosVariables.appendText ( infos.langue + "\n" );

            } else infosVariables.appendText ( "Les variables ne sont pas
disponibles" );

        }

    }

}
```

```
    }  
    }  
}
```

A la réception des variables la méthode écouteur `infos` est déclenchée et procède à un affichage des variables si celles-ci sont correctement définies.

## A retenir

- Afin d'assurer un accès simplifié aux variables, il est intéressant de diffuser un événement depuis le SWF principal aux SWF chargés.

## Appeler une fonction

La communication entre l'application conteneur et le lecteur ne se limite pas à un simple passage de variables encodées URL.

Dans le cas d'une page HTML conteneur, il est possible d'appeler une fonction JavaScript depuis ActionScript à l'aide de la fonction `navigateToURL`.

Dans le code suivant, nous ouvrons une fenêtre `alert` au sein de la page conteneur lorsque nous cliquons sur la scène :

```
package org.bytearray.document  
{  
    import flash.external.ExternalInterface;  
    import flash.text.TextField;  
    import flash.events.MouseEvent;  
    import flash.net.navigateToURL;  
    import flash.net.URLRequest;  
    import org.bytearray.abstrait.ApplicationDefault;  
  
    public class Document extends ApplicationDefault  
    {  
        public function Document ()  
        {  
            stage.addEventListener ( MouseEvent.CLICK, click );  
        }  
  
        private function click ( pEvt:MouseEvent ):void  
        {  
            navigateToURL ( new URLRequest ("javascript: alert('Un message du  
lecteur Flash !')"), "_self");  
        }  
    }  
}
```

```
    }  
  }  
}
```

En préfixant le nom de la fonction de l'instruction `javascript:` il est aussi possible d'appeler n'importe quelle fonction JavaScript depuis `ActionScript`.

Dans le code suivant, nous appelons une fonction `fonctionJavaScript`:

```
navigateToURL ( new URLRequest ("javascript: fonctionJavaScript()" ),  
              "_self");
```

La fonction `navigateToURL` est généralement utilisée au sein d'une page conteneur afin de lancer le gestionnaire de mail configuré sur la machine :

```
private function click ( pEvt:MouseEvent ):void  
{  
    navigateToURL ( new URLRequest ("mailto:bob@groove.com") );  
}
```

Bien qu'efficace, cette technique ne permet pas de réceptionner le retour d'une fonction JavaScript ou de passer des types précis en paramètres tels `Number` ou `Boolean`.

## L'API ExternalInterface

La communication entre l'application conteneur et le lecteur ne se limite pas à un simple passage de variables encodées URL.

Il est aussi possible d'appeler différentes méthodes définies au sein du conteneur depuis `ActionScript` et inversement. Cette fonctionnalité était assurée auparavant par la méthode `fscommand` qui se trouve désormais dans le paquetage `flash.system`.

L'utilisation de la fonction `fscommand` est aujourd'hui dépréciée au profit de l'API `ExternalInterface`.

Afin de pouvoir utiliser celle-ci dans un contexte de navigateur, celui-ci doit prendre en charge les contrôles `ActiveX` ou l'API `NPRuntime`.

Voici un tableau récapitulatif des différents navigateurs compatible fonctionnant avec l'API `ExternalInterface` :

Navigateur	Système d'exploitation	Système d'exploitation
------------	---------------------------	---------------------------



Internet Explorer 5.0 et versions ultérieures	Windows	
Netscape 8.0 et versions ultérieures	Windows	Macintosh
Mozilla 1.7.5 et versions ultérieures	Windows	Macintosh
Firefox 1.0 et versions ultérieures	Windows	Macintosh
Safari 1.3 et versions ultérieures		Macintosh

*Tableau 1. Navigateurs compatibles.*

Voici en détail les trois propriétés de la classe

`ExternalInterface` :

- `available` : Renvoie l'id de la balise `object` sous Internet Explorer, ou l'attribut `name` de la balise `embed` sous Netscape, Firefox, Opera ou autres.
- `marshallExceptions` : Cette propriété définit, si les deux acteurs peuvent recevoir les exceptions de chacun.
- `objectID` : Permet de savoir si le conteneur est compatible avec l'API `ExternalInterface`.

Dans le code suivant, nous testons si l'application évolue dans un contexte compatible avec l'API `ExternalInterface` :

```
package org.bytearray.document

{

    import flash.external.ExternalInterface;
    import flash.text.TextField;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault

    {

        public var compatible:TextField;

        public function Document ()

        {

            compatible.text = String ( ExternalInterface.available );

        }

    }

}
```

Il convient de toujours tester si le contexte actuel permet l'utilisation de l'API `ExternalInterface`. Pour cela, nous testons la propriété `available` avant toute tentative de communication.

La propriété `objectID` peut être aussi utilisée afin de déterminer si l'application évolue au sein du navigateur ou non.

Ainsi nous pourrions ajouter une propriété `navigateur` à la classe `ApplicationDefault` permettant de savoir si l'animation évolue au sein du navigateur ou non :

```
package org.bytearray.abstrait
{
    import flash.display.MovieClip;
    import flash.events.Event;
    import flash.display.Stage;
    import flash.external.ExternalInterface;

    public class ApplicationDefault extends MovieClip
    {
        public static var globalStage:Stage;
        public static var enLigne:Boolean;
        public static var navigateur:Boolean;
        public static var stage:Stage;
        public static var root:ApplicationDefault;

        public function ApplicationDefault ()
        {
            ApplicationDefault.root = this;

            addEventListener ( Event.ADDED_TO_STAGE, activation );

            loaderInfo.addEventListener ( Event.INIT, init );

            ApplicationDefault.navigateur = ExternalInterface.objectID !=
null;
        }

        private function activation ( pEvt:Event ):void
        {
            ApplicationDefault.globalStage = stage;
        }

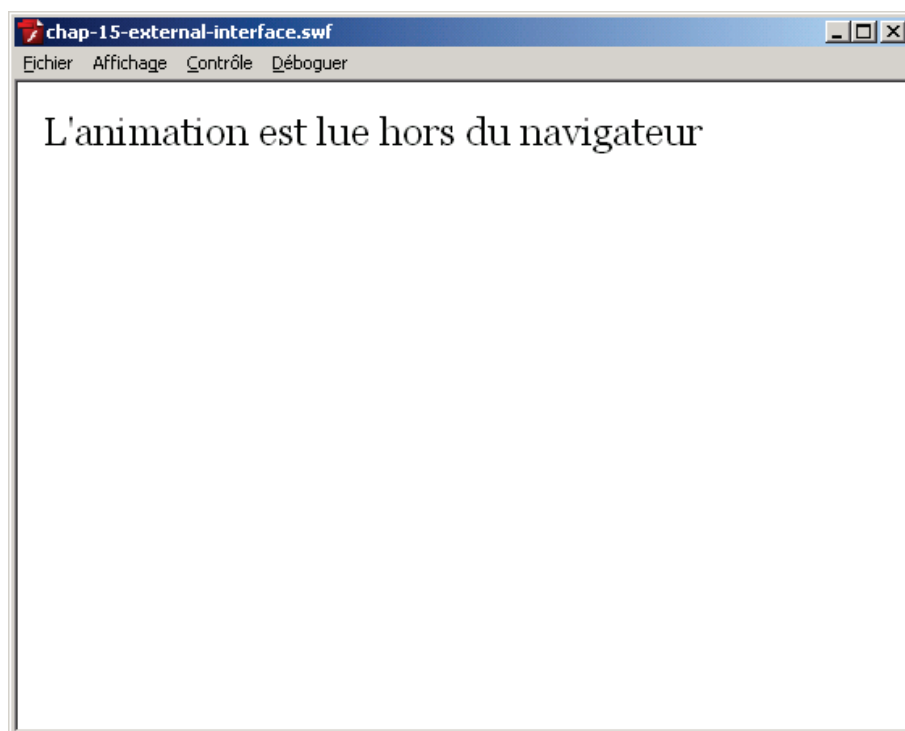
        private function init ( pEvt:Event ):void
        {
            ApplicationDefault.enLigne = pEvt.target.url.match ( new RegExp
            ("^http://") ) != null;
        }
    }
}
```

```
    }  
  }  
}
```

Puis, dans n'importe quelle classe de document héritant de la classe `ApplicationDefault`, nous pouvons savoir facilement si l'animation évolue ou non au sein du navigateur :

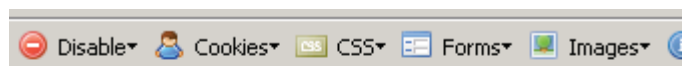
```
package org.bytearray.document  
  
{  
    import flash.text.TextField;  
    import org.bytearray.abstrait.ApplicationDefault;  
  
    public class Document extends ApplicationDefault  
    {  
        public var compatible:TextField;  
  
        public function Document ()  
        {  
            if ( ApplicationDefault.navigateur ) compatible.text =  
                "L'animation est lue au sein d'une page web";  
  
            else compatible.text = "L'animation est lue hors du navigateur";  
        }  
    }  
}
```

Si nous testons l'application hors du navigateur :



*Figure 15-7. Détection du contexte.*

A l'inverse, si l'animation est lue au sein du navigateur, nous le détectons comme l'illustre la figure 15-8 :



L'animation est lue au sein d'une page web

*Figure 15-8. Détection du contexte.*

Nous allons à présent nous attarder sur l'appel de fonctions externe depuis ActionScript.

## A retenir

- L'API `ExternalInterface` est recommandée pour la communication entre ActionScript et JavaScript.
- `ExternalInterface` remplace les précédentes fonctions `fscommand`, `callFrame` et `callLabel`.

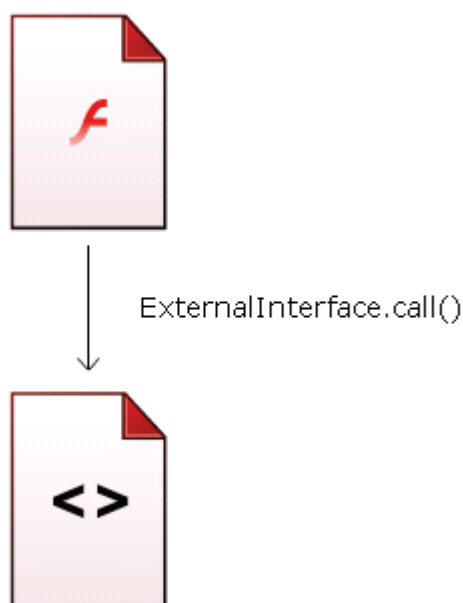
## Appeler une fonction externe depuis ActionScript

Deux méthodes sont disponibles sur la classe `ExternalInterface`, voici le détail de chacune d'entre elles :

- `addCallBack` : Enregistre un alias pour une fonction `ActionScript`. L'alias est ensuite utilisé depuis la fonction externe pour exécuter la fonction `ActionScript`.
- `call` : Exécute la fonction passée en paramètre au sein du conteneur.

Dans la partie suivante, nous allons nous intéresser à l'appel d'une méthode JavaScript depuis `ActionScript`.

La figure 15-9 illustre le concept :



*Figure 15-09. Méthode statique `call`.*

Afin d'exécuter une méthode au sein de l'application conteneur, nous utilisons la méthode statique `call` de la classe `ExternalInterface` dont voici la signature :

```
public static function call(functionName:String, ... arguments):*
```

Le premier paramètre concerne le nom de la fonction à exécuter. Les paramètres suivants sont passés en paramètre à la fonction exécutée.

Nous allons commencer par un exemple simple, en appelant la méthode `direBonjour` lorsque nous cliquons sur le bouton `executeFonction` :

```
package org.bytearray.document

{

    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.external.ExternalInterface;
```

```
import org.bytearray.abstrait.ApplicationDefault;

public class Document extends ApplicationDefault
{
    public var executeFonction:SimpleButton;

    public function Document ()
    {
        executeFonction.addEventListener ( MouseEvent.CLICK,
declencheAppel );

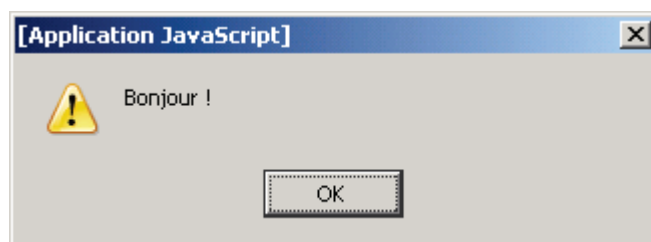
    }

    private function declencheAppel ( pEvt:MouseEvent ):void
    {
        if ( ExternalInterface.available )
        {
            ExternalInterface.call ("direBonjour");
        }
    }
}
}
```

La fonction JavaScript `direBonjour` est définie au sein de notre page conteneur :

```
function direBonjour ( )
{
    alert ("Bonjour !");
}
```

Lorsque nous cliquons sur le bouton, la fonction est déclenchée et ouvre une fenêtre d’alerte comme l’illustre la figure 15-9 :



*Figure 15-10. Détection du contexte.*

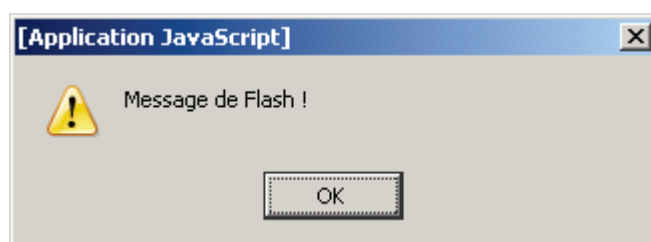
Nous pouvons passer dynamiquement des paramètres depuis Flash en spécifiant les paramètres à la méthode `call` :

```
ExternalInterface.call ("direBonjour", "Message de Flash !");
```

Nous modifions la fonction `direBonjour` afin d'accepter un message en paramètre :

```
function direBonjour ( pMessage )
{
    alert (pMessage);
}
```

La figure 15-11 illustre le résultat :



*Figure 15-11. Détection du contexte.*

La fonction JavaScript peut retourner une valeur, la valeur sera récupérée à l'appel de la méthode `call`.

Si nous modifions la fonction JavaScript afin que celle-ci renvoie le total des deux paramètres passés :

```
function calculTotal ( p1, p2 )
{
    return p1 + p2;
}
```

Nous affichons dans le champ texte `total`, le retour de l'appel de la méthode `call` :

```
package org.bytearray.document
{
    import flash.display.SimpleButton;
    import flash.events.MouseEvent;
    import flash.external.ExternalInterface;
    import flash.text.TextField;
    import org.bytearray.abstrait.ApplicationDefaut;

    public class Document extends ApplicationDefaut
    {
```

```
public var executeFonction:SimpleButton;
public var total:TextField;

public function Document ()
{
    executeFonction.addEventListener ( MouseEvent.CLICK,
declencheAppel );
}

private function declencheAppel ( pEvt:MouseEvent ):void
{
    if ( ExternalInterface.available )
    {
        total.text = ExternalInterface.call ("calculTotal", 10, 15);
    }
}
}
```

La figure 15-12 illustre le résultat :



25

*Figure 15-12. Détection du contexte.*

Le code précédent illustre un des avantages de l'API `ExternalInterface` concernant le passage de données typées.

Contrairement à la fonction `fscommand` ou `navigateToURL`, nous ne sommes pas à limités au passage de chaînes de caractères avec l'API `ExternalInterface`. Nous pouvons passer entre JavaScript et ActionScript des données de type `Number`, `String`, et `Boolean`.

Nous allons maintenant communiquer dans l'autre sens en appelant depuis l'application conteneur une fonction ActionScript.

## A retenir



- La méthode `call` permet d'exécuter une fonction définie au sein du conteneur depuis `ActionScript`.
- Dans le cas de l'utilisation d'une fonction JavaScript, des types comme `Number`, `Boolean` et `String` peuvent être échangés.

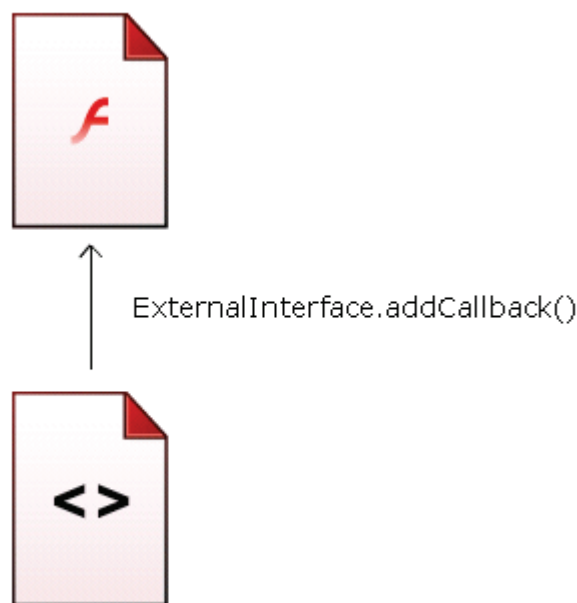
## Appeler une fonction `ActionScript` depuis le conteneur

De la même manière, nous pouvons déclencher une fonction `ActionScript` depuis une fonction définie au sein du conteneur.

Afin d'enregistrer une fonction `ActionScript` auprès d'une fonction du conteneur, nous utilisons la méthode `addCallback`, dont voici la signature :

```
public static function addCallback(functionName:String,  
closure:Function):void
```

La figure 15-13 illustre l'idée :



*Figure 15-13. Méthode statique `addCallback`.*

Voici le détail des deux paramètres de la méthode `addCallback` :

- `functionName` : Il s'agit de l'identifiant de la fonction depuis la page conteneur. Nous devons utiliser cette chaîne pour exécuter la fonction passée au sein du paramètre `closure`.
- `closure` : La fonction à déclencher depuis la page conteneur.

Dans le code suivant, nous enregistrons une fonction `maFunction` à l'aide de l'alias `aliasFonction` :

```
package org.bytearray.document

{

    import flash.external.ExternalInterface;
    import flash.text.TextField;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault

    {

        public var messageJavascript:TextField;

        public function Document ()

        {

            // enregistre l'alias "aliasFonction" vers la méthode maFunction
            ExternalInterface.addCallback ( "aliasFonction", maFunction );

        }

        private function maFunction ( pMessage:String ):void

        {

            // nous affichons le message reçu depuis JavaScript
            messageJavascript.text = pMessage;

        }

    }

}
```

Afin de pouvoir appeler une fonction ActionScript nous devons ensuite donner un nom à notre application grâce aux attributs `id` et `name` :

```
AC_FL_RunContent(
    'codebase',
    'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=
    9,0,0,0',
    'width', '550',
    'height', '400',
    'src', 'chap-15-external-interface',
    'quality', 'high',
    'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
    'align', 'middle',
    'play', 'true',
    'loop', 'true',
    'scale', 'showall',
    'wmode', 'window',
    'devicefont', 'false',
    'id', 'monApplication',
    'bgcolor', '#ffffff',
    'name', 'monApplication',
    'menu', 'true',
    'allowFullScreen', 'false',
    'allowScriptAccess', 'sameDomain',
    'movie', 'chap-15-external-interface',
```

```
'salign', ''  
);
```

Puis au sein du conteneur, nous appelons la fonction `maFunction` grâce à l'alias `aliasFunction` :

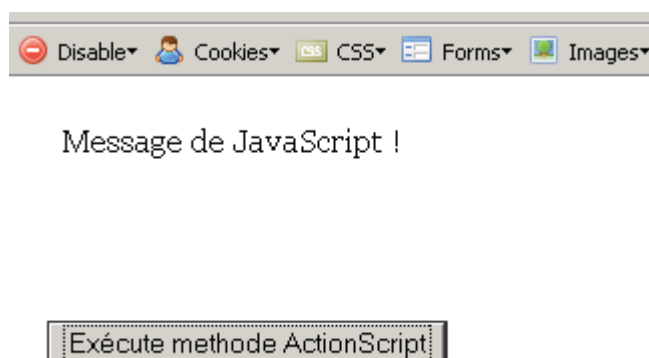
```
function recupAnimation ( pAnim )  
{  
  
    if (navigator.appName.indexOf("Microsoft") != -1) return window[pAnim];  
    else return document[pAnim];  
  
}  
  
function declencheFonctionActionScript ( )  
{  
  
    recupAnimation("monApplication").aliasFunction("Message de JavaScript !");  
  
}
```

La fonction `recupAnimation` permet de cibler l'animation selon le type de navigateur utilisé. Nous passons le nom de l'animation à celle-ci, qui nous renvoie l'animation intégrée dans la page, sur laquelle nous appelons la fonction grâce à l'alias passé à la méthode `addCallback`.

Il ne nous reste plus qu'à déclencher la fonction `declencheFonctionActionScript` lors du clic bouton :

```
<input type="button" name="monBouton" value="Exécute fonction ActionScript"  
onClick=" declencheFonctionActionScript()">
```

La figure 15-14 illustre le résultat :



*Figure 15-14. Méthode ActionScript exécutée.*

Comme vous pouvez l'imaginer, la notion de communication entre le lecteur Flash et l'application conteneur est soumise à des restrictions de sécurité.

Nous allons nous intéresser dans la partie suivante, aux différentes restrictions possibles.

## A retenir

- La méthode `addCallback` permet d'exécuter une fonction ActionScript depuis l'application conteneur.
- Dans le cas de l'utilisation d'une fonction JavaScript, des types comme `Number`, `Boolean` et `String` peuvent être échangés.

## Communication et sécurité

Afin que les deux acteurs puissent communiquer, nous devons nous intéresser à la valeur passée à l'attribut `allowScriptAccess`.

La fonction `AC_FL_RunContent` intègre un paramètre `allowScriptAccess` régissant la communication entre l'application conteneur et le lecteur Flash :

```
AC_FL_RunContent(
    'codebase',
    'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=
    9,0,0,0',
    'width', '550',
    'height', '400',
    'src', 'chap-15-external-interface',
    'quality', 'high',
    'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
    'align', 'middle',
    'play', 'true',
    'loop', 'true',
    'scale', 'showall',
    'wmode', 'window',
    'devicefont', 'false',
    'id', 'monApplication',
    'bgcolor', '#ffffff',
    'name', 'monApplication',
    'menu', 'true',
    'allowFullScreen', 'false',
    'allowScriptAccess', 'sameDomain',
    'movie', 'chap-15-external-interface',
    'salign', ''
);
```

Voici les trois valeurs possibles de l'attribut `allowScriptAccess` :

- `always` : La communication entre le l'application conteneur et ActionScript est toujours possible.
- `sameDomain` : La communication entre l'application conteneur et ActionScript est possible, uniquement si la page conteneur et le SWF évoluent dans le même domaine.
- `never` : La communication entre le l'application conteneur et ActionScript est impossible.

Nous pourrions nous demander dans quels cas l'utilisation de l'attribut `allowScriptAccess` serait nécessaire.

Imaginons le scénario suivant :

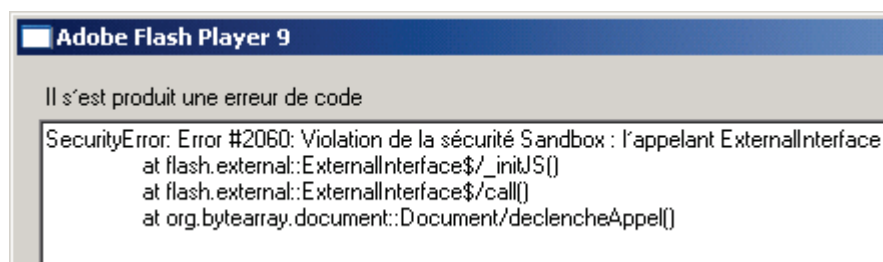
Au sein d'un forum, les utilisateurs ont la possibilité d'intégrer leurs avatars ou signature à partir d'animations Flash. Certaines d'entre elles pourraient scripter la page du forum provoquant alors un dysfonctionnement.

Afin de réguler cela, nous pourrions passer la valeur `never` à l'attribut `allowScriptAccess` empêchant toute communication entre les pages du forum et les avatars ou signatures.

Dans le code suivant, nous passons l'attribut `allowScriptAccess` à `never` :

```
AC_FL_RunContent(
    'codebase',
    'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=
    9,0,0,0',
    'width', '550',
    'height', '400',
    'src', 'chap-15-external-interface',
    'quality', 'high',
    'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
    'align', 'middle',
    'play', 'true',
    'loop', 'true',
    'scale', 'showall',
    'wmode', 'window',
    'devicefont', 'false',
    'id', 'monApplication',
    'bgcolor', '#ffffff',
    'name', 'monApplication',
    'menu', 'true',
    'allowFullScreen', 'false',
    'allowScriptAccess', 'never',
    'movie', 'chap-15-external-interface',
    'salign', ''
);
```

Si nous tentons d'appeler à l'aide de la méthode `call` une fonction définie au sein de la page conteneur, une exception de type `SecurityError` est levée.



*Figure 15-15. Restriction de sécurité.*

A l'inverse, pour que la page conteneur puisse scripter un SWF provenant d'un domaine différent, nous pouvons utiliser la méthode `allowDomain` de la classe `Security` au sein du SWF à scripter.

Pour plus d'informations concernant le modèle de sécurité du lecteur Flash, reportez vous à la partie *Modèle de sécurité du lecteur Flash* du chapitre 13 intitulé *Chargement de contenu*.

### A retenir

- La communication entre le lecteur Flash et l'application conteneur est soumise au modèle de sécurité du lecteur Flash.
- L'attribut `allowScriptAccess` permet d'autoriser ou non le lecteur Flash à scripter la page conteneur.

# 16

## Le texte

<b>LE TEXTE DANS FLASH.....</b>	<b>1</b>
AFFICHER DU TEXTE DANS FLASH.....	2
AFFICHER DU TEXTE EN PROGRAMMANT.....	5
LES TYPES DE CHAMP TEXTE .....	13
<b>FORMATAGE DU TEXTE .....</b>	<b>15</b>
RENDU HTML .....	15
LA CLASSE TEXTFORMAT .....	17
ETENDRE LA CLASSE TEXTFIELD .....	24
LA CLASSE STYLE SHEET .....	30
<b>MODIFIER LE CONTENU D’UN CHAMP TEXTE .....</b>	<b>34</b>
REEMPLACER DU TEXTE.....	37
<b>L’EVENEMENT TEXTEVENT.LINK .....</b>	<b>39</b>
<b>CHARGER DU CONTENU EXTERNE .....</b>	<b>44</b>
<b>EXPORTER UNE POLICE DANS L’ANIMATION .....</b>	<b>49</b>
CHARGER DYNAMIQUEMENT UNE POLICE .....	53
<b>DETECTER LES COORDONNEES DE TEXTE .....</b>	<b>57</b>
<b>CREER UN EDITEUR DE TEXTE .....</b>	<b>68</b>

### Le texte dans Flash

La gestion du texte dans le lecteur Flash a souvent été source de discussions au sein de la communauté. Quelque soit le type d’application produite, l’utilisation du texte s’avère indispensable.

Le lecteur Flash 8 a permis l’introduction d’un nouveau moteur de rendu du texte appelé *Saffron*. Ce dernier améliorait nettement la lisibilité du texte dans une taille réduite et offrait un meilleur contrôle sur le niveau de lissage des polices.

ActionScript 3 n'intègre pas de nouveau moteur de rendu, mais enrichit remarquablement les fonctionnalités offertes par la classe `TextField` et ouvre de nouvelles possibilités.

Trois classes permettent de représenter du texte en ActionScript 3 :

- `flash.text.StaticText` : la classe `StaticText` représente les champs texte statiques créés depuis l'environnement auteur ;
- `flash.text.TextField` : la classe `TextField` représente les champs texte dynamiques créés depuis l'environnement auteur ainsi que la création de champs texte par programmation ;
- `flash.text.TextSnapshot` : la classe `TextSnapshot` permet de travailler avec le contenu d'un champ texte statique créé depuis l'environnement auteur.

Bien que ces classes ne résident pas dans le paquetage `flash.display`, celles-ci héritent de la classe `DisplayObject`.

Nous allons nous intéresser au cours de ce chapitre à quelques cas pratiques d'utilisation du texte qui auraient été difficilement réalisables sans l'utilisation d'ActionScript 3.

## Afficher du texte dans Flash

Afin de nous familiariser avec le texte, nous allons créer un simple champ texte statique au sein d'un nouveau document comme l'illustre la figure 16-1 :



*Figure 16-1. Champ texte statique.*

Il est impossible d'attribuer un nom d'occurrence à un champ texte statique, ActionScript 3 nous permet cependant d'accéder au champ texte en utilisant la méthode `getChildAt` :

```
// affiche : [Object StaticText]
trace( getChildAt ( 0 ) );
```

Lorsqu'un champ texte statique est créé depuis l'environnement auteur, celui-ci est de type `StaticText`. Cette classe définit une seule et unique propriété `text` en lecture seule permettant de récupérer le contenu du champ :

```
var champTexte:StaticText = StaticText ( getChildAt ( 0 ) );
```



```
// affiche : Contenu statique  
trace( champTexte.text );
```

Si nous tentons d’instancier la classe `StaticText` :

```
var monChamp:StaticText = new StaticText();
```

Une erreur à l’exécution de type `ArgumentError` est levée :

```
ArgumentError: Error #2012: Impossible d'instancier la classe StaticText
```

Il est important de noter que seul l’environnement auteur de Flash CS3 permet la création d’instance de la classe `StaticText`.

Bien que celle-ci ne définisse qu’une seule propriété, la classe `StaticText` hérite de la classe `DisplayObject`. Il est donc possible de modifier la présentation du champ.

Dans le code suivant nous modifions la rotation du texte, ainsi que l’opacité du champ :

```
var champTexte:StaticText = StaticText ( getChildAt ( 0 ) );  
  
champTexte.rotation = 20;  
champTexte.alpha = .5;
```

La figure 16-2 illustre le résultat :



*Figure 16-2. Présentation du champ texte statique.*

Dans le cas de champs texte dynamiques, assurez-vous d’avoir bien intégré les contours de police afin de pouvoir faire subir une rotation au texte sans que celui-ci ne disparaisse. Nous reviendrons très bientôt sur cette notion.

Dans les précédentes versions d’ActionScript il était impossible d’accéder ou de modifier directement la présentation d’un champ texte statique créé depuis l’environnement auteur.

En modifiant le type du champ, en texte dynamique ou de saisie, nous pouvons lui donner un nom d’occurrence comme l’illustre la figure 16-3 :



*Figure 16-3. Champ texte dynamique.*

A la compilation, une variable du même nom est créée afin de référencer notre champ texte.

Nous ciblons ce dernier à travers le code suivant :

```
// affiche : [object TextField]
trace( legende );
```

Nous pouvons remarquer que le type du champ n'est plus `StaticText` mais `TextField`.

Contrairement à la classe `StaticText`, les différentes propriétés et méthodes définies par la classe `TextField` permettent une manipulation avancée du texte :

```
legende.text = "Nouveau contenu";

// affiche : 56
trace( legende.textWidth );

// affiche : 0
trace( legende.textColor );

// affiche : dynamic
trace( legende.type );
```

Afin de continuer notre exploration du texte au sein d'ActionScript 3, nous allons nous intéresser à présent à la création de texte dynamique.

## A retenir

- Trois classes peuvent être utilisées afin de manipuler du texte : `StaticText`, `TextField`, `TextSnapshot`.
- ActionScript 3 n'intègre pas de nouveau moteur de rendu mais enrichit les capacités de la classe `TextField`.
- Les champs texte statique sont de type `StaticText`.
- Seul l'environnement auteur de Flash CS3 permet la création de champ texte de type `StaticText`.
- Il est impossible d'instancier manuellement la classe `StaticText`.
- Les champs texte dynamique et de saisie sont de type `TextField`.

## Afficher du texte en programmant

Lorsque nous devons afficher du texte par programmation, nous utilisons la classe `flash.text.TextField`. Nous allons pour démarrer, créer un simple champ texte et y ajouter du contenu.

La classe `TextField` s'instancie comme tout autre `DisplayObject` à l'aide du mot clé `new` :

```
var monTexte:TextField = new TextField();  
  
monTexte.text = "Bonjour !";
```

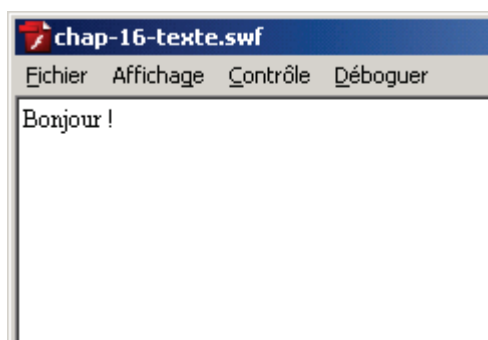
Il est important de noter que grâce au nouveau mode d'instanciation des `DisplayObject`, il n'est plus nécessaire de disposer d'une instance de `MovieClip` afin de créer un champ texte.

Rappelez-vous, la méthode `createTextField` présente en ActionScript et 2 était définie sur la classe `MovieClip`. Il était donc impossible de créer un champ texte sans clip d'animation.

Une fois le champ texte créé nous pouvons l'afficher en l'ajoutant à la liste d'affichage :

```
var monTexte:TextField = new TextField();  
  
monTexte.text = "Bonjour !";  
  
addChild ( monTexte );
```

Si nous testons le code précédent nous obtenons le résultat illustré en figure 16-4 :



*Figure 16-4. Contenu texte.*

La propriété `text` de l'objet `TextField` nous permet d'affecter du contenu, nous allons découvrir très vite de nouvelles propriétés et méthodes liées à l'affectation de contenu.

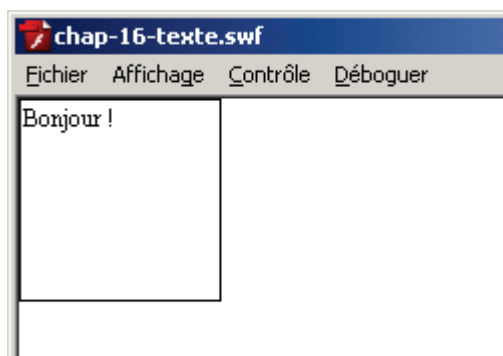
Par défaut, la taille d'un champ texte créé par programmation est de 100 pixels en largeur et en hauteur :

```
var monTexte:TextField = new TextField();  
monTexte.text = "Bonjour !";  
addChild ( monTexte );  
// affiche : 100 100  
trace( monTexte.width, monTexte.height );
```

Afin de vérifier cela visuellement, nous pouvons activer la propriété `border` de l'objet `TextField` qui ajoute un contour en bordure du champ texte :

```
var monTexte:TextField = new TextField();  
monTexte.border = true;  
monTexte.text = "Bonjour !";  
addChild ( monTexte );
```

La figure 16-5 illustre les dimensions d'un champ texte par défaut :

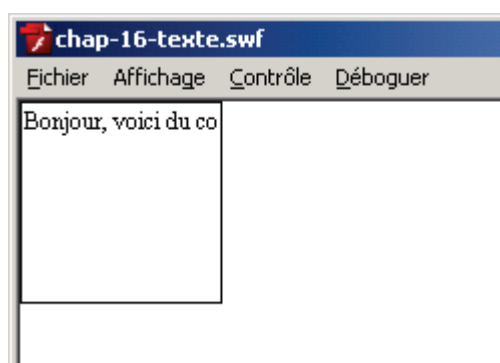


*Figure 16-5. Dimensions par défaut du champ texte.*

Si le contenu dépasse cette surface par défaut, le texte est tronqué. Dans le code suivant nous modifions le contenu du champ afin de le faire déborder :

```
var monTexte:TextField = new TextField();  
  
monTexte.border = true;  
  
monTexte.text = "Bonjour, voici du contenu qui déborde !";  
  
addChild ( monTexte );
```

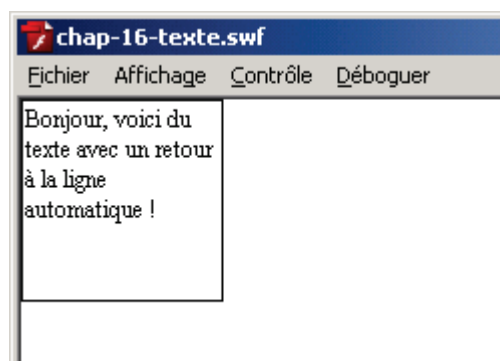
Le texte est alors tronqué comme l'illustre la figure 16-6 :

*Figure 16-6. Texte tronqué.*

Afin que le texte revienne à la ligne automatiquement nous activons la propriété `wordWrap` qui par défaut est définie à `false` :

```
var monTexte:TextField = new TextField();  
  
monTexte.border = true;  
  
monTexte.text = "Bonjour, voici du texte avec un retour à la ligne  
automatique !";  
  
monTexte.wordWrap = true;  
  
addChild ( monTexte );
```

La figure 16-7 illustre le résultat :



*Figure 16-7. Texte contraint avec retour à la ligne automatique.*

A l'inverse si nous souhaitons que le champ s'adapte automatiquement au contenu, nous utilisons la propriété `autoSize`. En plus d'assurer un redimensionnement automatique, celle-ci permet de justifier le texte dans différentes positions.

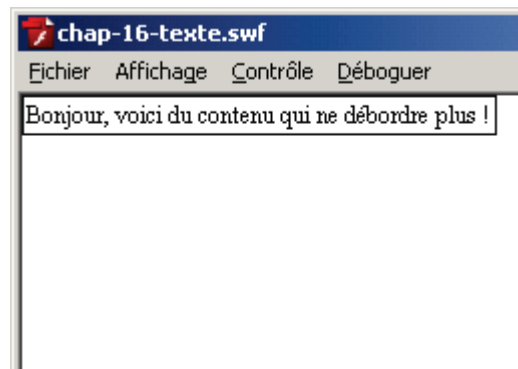
La propriété `autoSize` prend comme valeur, une des quatre propriétés constantes de la classe `TextFieldAutoSize` :

- `TextFieldAutoSize.CENTER` : le texte est justifié au centre ;
- `TextFieldAutoSize.LEFT` : le texte est justifié à gauche ;
- `TextFieldAutoSize.NONE` : aucune justification du texte ;
- `TextFieldAutoSize.RIGHT` : le texte est justifié à droite ;

Dans le code suivant nous justifions le texte à gauche du champ :

```
var monTexte:TextField = new TextField();  
monTexte.border = true;  
monTexte.text = "Bonjour, voici du contenu qui ne débordre plus !";  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
addChild ( monTexte );
```

La figure 16-8 illustre le résultat :



*Figure 16-8. Redimensionnement automatique du champ.*

Cette fonctionnalité est couramment utilisée pour la mise en forme du texte au sein de paragraphes. Nous l'utiliserons lorsque nous développerons un éditeur de texte à la fin de ce chapitre.

Si nous souhaitons ajouter une contrainte de dimensions afin de contenir le texte au sein d'un bloc spécifique, nous pouvons affecter au champ une largeur et une hauteur spécifique. Un retour à la ligne

automatique devra être ajouté, tout en s’assurant de ne pas spécifier de redimensionnement par la propriété `autoSize`.

Pour cela nous utilisons les propriétés `width` et `height` et `wordWrap` de l’objet `TextField` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Le texte évolue dans un paragraphe d’une largeur et hauteur  
de 250 pixels maximum. Si le texte dépasse, un retour à la ligne est ajouté  
automatiquement.";

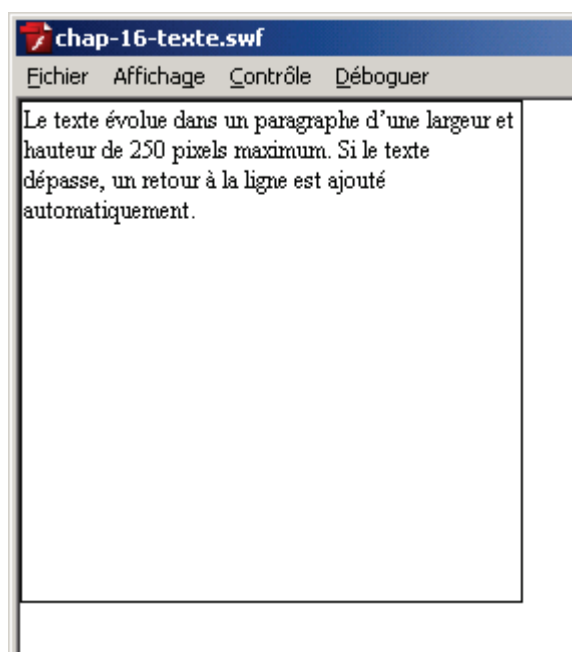
monTexte.width = 250;
monTexte.height = 250;

monTexte.wordWrap = true;

addChild ( monTexte );
```

Dans le code précédent, le texte est contraint à un paragraphe d’une largeur et hauteur de 250 pixels maximum.

La figure 16-9 illustre le bloc de texte :



*Figure 16-9. Retour à la ligne automatique.*

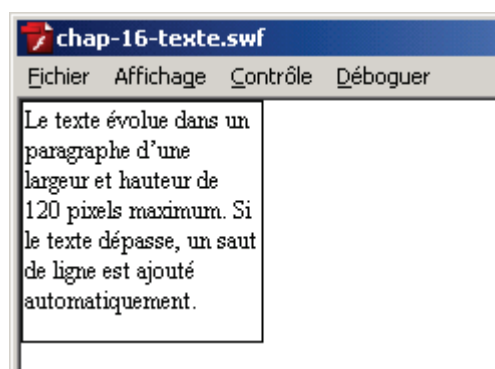
Si nous réduisons la largeur du champ texte, le texte est alors contraint d’évoluer dans cette surface :

```
var monTexte:TextField = new TextField();

monTexte.border = true;
```

```
monTexte.text = "Le texte évolue dans un paragraphe d'une largeur et hauteur  
de 120 pixels maximum. Si le texte dépasse, un saut de ligne est ajouté  
automatiquement.";  
  
monTexte.width = 120;  
monTexte.height = 120;  
  
monTexte.wordWrap = true;  
  
addChild ( monTexte );
```

La figure 16-10 illustre le résultat :



*Figure 16-10. Champ texte restreint.*

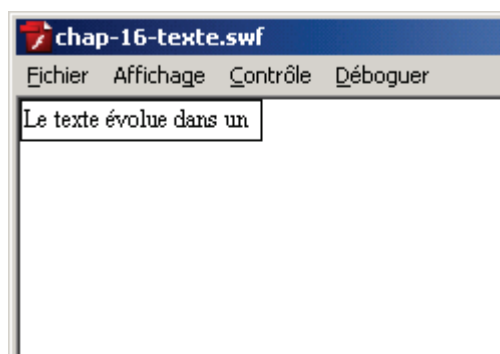
Quelles que soient les dimensions affectées au champ texte, si nous activons le redimensionnement automatique, le lecteur Flash affichera totalement le texte, quitte à étendre la hauteur du champ.

Dans le code suivant, nous réduisons les dimensions du champ texte :

```
var monTexte:TextField = new TextField();  
  
monTexte.border = true;  
  
monTexte.text = "Le texte évolue dans un paragraphe d'une largeur et hauteur  
de 120 pixels maximum. Si le texte dépasse, un saut de ligne est ajouté  
automatiquement.";  
  
monTexte.width = 120;  
monTexte.height = 30;  
  
monTexte.wordWrap = true;  
  
addChild ( monTexte );
```

La figure 16-11 illustre le rendu :





*Figure 16-11. Texte tronqué.*

Si nous ajoutons un ajustement automatique par la propriété `autoSize`, le lecteur Flash conserve une largeur maximale de 120 pixels mais augmente la hauteur du champ afin d'afficher le contenu total :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.text = "Le texte évolue dans un paragraphe d'une largeur et hauteur
de 120 pixels maximum. Si le texte dépasse, un saut de ligne est ajouté
automatiquement.";

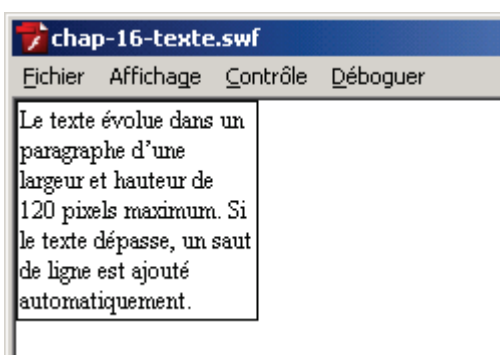
monTexte.width = 120;
monTexte.height = 30;

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.wordWrap = true;

addChild ( monTexte );
```

La figure 16-12 illustre le comportement :



*Figure 16-12. Redimensionnement automatique.*

Nous pouvons dynamiquement modifier les dimensions du champ afin de comprendre comment le lecteur ajuste le contenu du champ :

```
var monTexte:TextField = new TextField();
```

```

monTexte.border = true;

monTexte.text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Pellentesque at metus a magna bibendum semper. Suspendisse id mauris. Duis
consequat dolor et odio. Integer euismod enim ut nulla. Sed quam. Cum sociis
natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.
Integer lobortis. In non erat. Sed ac dui a arcu ultrices aliquam. Aenean
neque neque, vulputate ac, dictum eu, vestibulum ut, enim. Sed quis eros.
Curabitur eu odio ac nisi suscipit venenatis. Duis ultrices viverra sapien.
Fusce interdum, felis eget mollis varius, enim sem imperdiet leo, sed
sagittis turpis odio sed quam. Phasellus ac orci. Morbi vestibulum, sem at
cursus auctor, metus odio suscipit ipsum, ac sodales erat mi ac velit. Nullam
tempus iaculis sem.";

monTexte.wordWrap = true;

addChild ( monTexte );

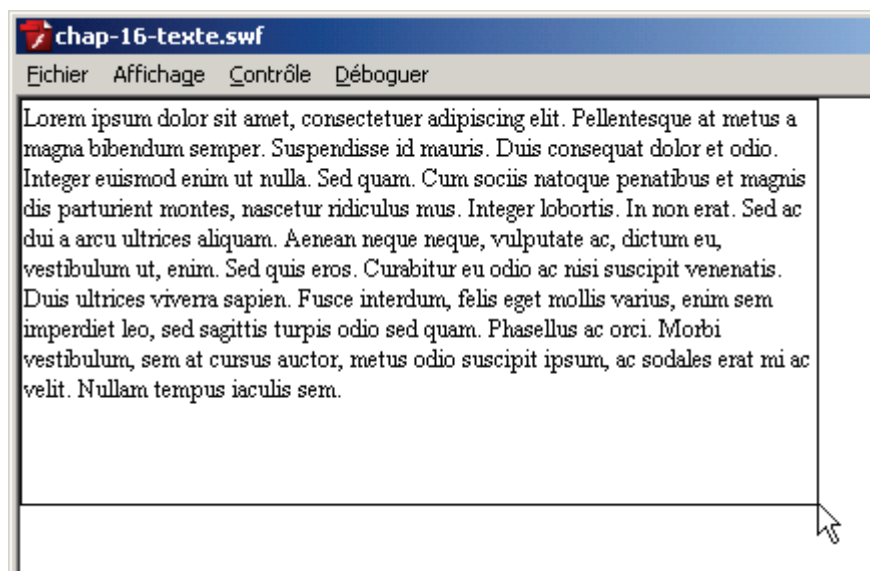
stage.addEventListener ( MouseEvent.MOUSE_MOVE, ajustement );

function ajustement ( pEvt:MouseEvent ):void
{
    monTexte.width = pEvt.stageX;
    monTexte.height = pEvt.stageY;

    pEvt.updateAfterEvent();
}

```

La figure 16-13 illustre le résultat :



*Figure 16-13. Taille du paragraphe dynamique.*

Nous utilisons la méthode `updateAfterEvent` afin de forcer le rafraîchissement du lecteur.

Lorsqu'un champ texte est créé, celui-ci est par défaut éditable permettant une sélection du texte. Si nous souhaitons rendre le champ non sélectionnable nous passons la propriété `selectable` à `false` :

```
monTexte.selectable = false;
```

Nous allons nous intéresser au cours de la prochaine partie aux différents types de champs texte dynamiques existants.

## A retenir

- Un champ texte créé par programmation possède une taille par défaut de 100 pixels en largeur et hauteur.
- Les propriétés `width` et `height` permettent de définir la taille d'un champ texte.
- La propriété `wordWrap` permet d'activer le retour à la ligne automatique.
- Afin d'adapter le champ texte dynamique au contenu, nous utilisons la propriété `autoSize` et l'une des quatres constantes de la classe `TextFieldAutoSize`.
- La propriété `autoSize` permet aussi de définir l'ajustement du texte au sein du champ.
- Si un dimensionnement automatique a été spécifié et que le champ texte est trop petit, le lecteur Flash étend le champ dans le sens de la hauteur pour afficher la totalité du texte.

## Les types de champ texte

Deux comportements existent pour les champs texte dynamiques. Il est possible par la propriété `type` de récupérer ou de modifier le comportement du champ.

Deux types de champs texte dynamiques sont définis par la classe `TextFieldType` :

- `TextFieldType.DYNAMIC` : le champ texte est de type dynamique.
- `TextFieldType.INPUT` : le champ texte est de type saisie.

Par défaut, un champ texte créé à partir de la classe `TextField` est considéré comme `dynamic` :

```
var monTexte:TextField = new TextField();  
  
monTexte.border = true;  
  
monTexte.text = "Voici du contenu qui ne déborde plus !";  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
addChild ( monTexte );
```

```
// affiche : dynamic
trace( monTexte.type );

// affiche : true
trace( monTexte.type == TextFieldType.DYNAMIC );
```

Dans ce cas, celui ci peut être manipulé par programmation aussi bien au niveau de sa présentation qu’au niveau de son contenu.

Afin de créer un champ texte de saisie, nous modifions la propriété `type` et passons la chaîne `TextFieldType.INPUT` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

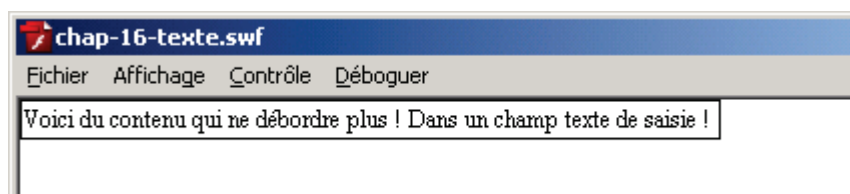
monTexte.type = TextFieldType.INPUT;

monTexte.text = "Voici du contenu qui ne débordre plus ! Dans un champ texte
de saisie !";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );
```

Le champ texte permet désormais la saisie du texte à la volée :



*Figure 16-14. Champ texte auto-adapté.*

Notons que dans le code précédent, nous ne pouvons sauter des lignes manuellement. Afin d’activer cette fonctionnalité, nous utilisons le mode multi-lignes à l’aide de la propriété `multiline` :

```
var monTexte:TextField = new TextField();

monTexte.border = true;

monTexte.multiline = true;

monTexte.type = TextFieldType.INPUT;

monTexte.text = "Bonjour, voici du contenu qui ne débordre plus !";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );
```

Dans la prochaine partie, nous allons nous intéresser aux différentes techniques permettant la modification du texte.

## Formatage du texte

ActionScript 3 dispose de trois techniques assurant le formatage du texte, voici un récapitulatif de chacune d'entre elles :

- Le texte HTML : de simples balises HTML peuvent être intégrées au contenu texte afin d'assurer son formatage.
- La classe `flash.text.TextFormat` : à l'aide d'un objet `TextFormat`, le texte peut être mis en forme aisément et rapidement.
- La classe `flash.text.StyleSheet` : la classe `StyleSheet` permet la création d'une feuille de style CSS, idéale pour la mise en forme de contenu conséquent.

Dans la mesure où nous affectons du contenu, le formatage du texte ne concerne que les champs texte dynamique ou de saisie. Rappelez vous qu'il est impossible de modifier le contenu d'un champ `StaticText` ou `TextSnapshot`.

## Rendu HTML

Nous allons nous intéresser dans un premier temps à une première technique de formatage, par balises HTML, à l'aide de la propriété `htmlText`.

---

Consultez l'aide de Flash CS3 pour la liste des balises HTML gérées par le lecteur Flash.

---

Dans le code suivant, nous créons un simple champ texte :

```
var monTexte:TextField = new TextField();  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
monTexte.text = "Voici du texte !";  
  
addChild ( monTexte );
```

Si nous accédons à la propriété `text`, le lecteur Flash renvoie la chaîne de caractères sans aucune information de formatage :

```
var monTexte:TextField = new TextField();  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
monTexte.text = "Voici du texte au format HTML";  
  
addChild ( monTexte );  
  
// affiche : Voici du texte au format HTML  
trace( monTexte.text );
```

A l'inverse, si nous accédons au contenu, par l'intermédiaire de la propriété `htmlText`, le lecteur renvoie le texte accompagné des différentes balises HTML utilisées en interne :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte au format HTML";

addChild ( monTexte );

/* affiche :
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">Voici du texte au format HTML</FONT></P>
*/
trace( monTexte.htmlText );
```

En modifiant le couleur du texte à l'aide de la propriété `textColor`, nous remarquons que l'attribut `color` de la balise `<font>` est automatiquement mis à jour :

```
var monTexte:TextField = new TextField();

monTexte.textColor = 0x990000;

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte au format HTML";

addChild ( monTexte );

/* affiche :
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#990000"
LETTERSPACING="0" KERNING="0">Voici du texte au format HTML</FONT></P>
*/
trace( monTexte.htmlText );
```

Nous affectons ainsi du contenu HTML de manière indirecte, par l'intermédiaire des différentes propriétés de la classe `TextField`.

En utilisant la propriété `htmlText`, les balises sont automatiquement interprétées :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "Voici du texte au format <b>HTML</b>";

addChild ( monTexte );
```

La figure 16-15 illustre le rendu :



Figure 16-15. Champ texte HTML formaté.

Notons que la propriété `html` de la classe `TextField` n'existe plus en ActionScript 3. L'affectation du texte par la propriété `htmlText` active automatiquement l'interprétation des balises HTML.

Nous pouvons changer la couleur du texte en utilisant l'attribut `color` de la balise `<font>` :

```
monTexte.htmlText = "Voici du texte <font color='#FF0000'>rouge</font> au  
format <b>HTML</b>";
```

L'affectation de contenu HTML s'avère extrêmement pratique dans la mesure où le contenu texte possède au préalable toutes les balises liées à sa présentation. En revanche, une fois le contenu affecté, il s'avère difficile de modifier la mise en forme du texte à l'aide des balises HTML. Pour cela, nous préférons généralement l'utilisation d'un objet `TextFormat`.

## La classe `TextFormat`

L'objet `TextFormat` a été conçu pour représenter la mise en forme du texte. Celui-ci doit être considéré comme un *outil de formatage*.

En réalité, l'objet `TextFormat` génère automatiquement les balises HTML appropriées afin de styliser le texte. La classe `TextFormat` bénéficie ainsi d'un grand nombre de propriétés permettant une mise en forme du texte avancée.

---

Consultez l'aide de Flash CS3 pour la liste des propriétés liées à la mise en forme définie par la classe `TextFormat`.

---

La première étape consiste à créer l'objet `TextFormat`, puis de définir les propriétés de mise en forme :

```
var monTexte:TextField = new TextField();  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
monTexte.text = "Voici du texte mis en forme";  
  
addChild ( monTexte );  
  
// création de l'objet TextFormat ( mise en forme )
```

```
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";
```

Afin d'appliquer cette mise en forme, nous devons à présent appeler la méthode `setTextFormat` de la classe `TextField` dont voici la signature :

```
public function setTextFormat(format:TextFormat, beginIndex:int = -1,
                              endIndex:int = -1):void
```

Analysons chacun des paramètres :

- `format` : l'objet `TextFormat` déterminant la mise en forme du texte.
- `beginIndex` : la position du caractère de départ auquel appliquer la mise en forme, la valeur par défaut est -1.
- `endIndex` : la position du caractère de fin auquel appliquer la mise en forme, la valeur par défaut est -1.

Dans le code suivant, nous appliquons la mise en forme à la totalité du champ texte :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte mis en forme";

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

// application de la mise en forme
monTexte.setTextFormat( miseEnForme );
```

La figure 16-16 illustre le résultat :





*Figure 16-16. Mise en forme du texte par l'objet  
TextFormat.*

En spécifiant les positions des caractères de début et fin, nous pouvons affecter le style à une partie du texte seulement :

```
// application de la mise en forme à une partie du texte
monTexte.setTextFormat( miseEnForme, 22, 27 );
```

Comme l'illustre la figure 16-17, seul le mot forme est affecté :



*Figure 16-17. Mise en forme partielle du texte.*

En interne, l'affectation de la mise en forme génère les balises HTML appropriées :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.text = "Voici du texte mis en forme";

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

/* affiche :
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"
LETTERSPACING="0" KERNING="0">Voici du texte au format HTML</FONT></P>
*/
trace( monTexte.htmlText );

// application de la mise en forme
monTexte.setTextFormat( miseEnForme, 22, 27 );
```

```
/* affiche :  
<P ALIGN="LEFT"><FONT FACE="Times New Roman" SIZE="12" COLOR="#000000"  
LETTERSPACING="0" KERNING="0">Voici du texte au format <FONT FACE="Verdana"  
SIZE="14"><B>HTML</B></FONT></FONT></P>  
*/  
trace( monTexte.htmlText );
```

Il est important de noter que lors de l'affectation de contenu par la propriété `text` ou `htmlText`, le lecteur crée automatiquement en interne un objet `TextFormat` associé.

---

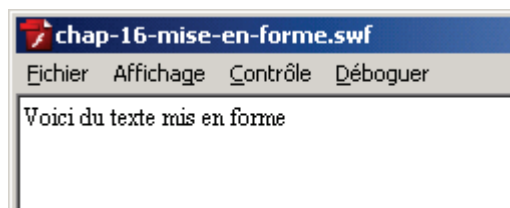
Il convient donc d'appeler la méthode `setTextFormat` *après* l'affectation du contenu, au risque de voir la mise en forme préalablement appliquée écrasée. En revanche, une fois la méthode `setTextFormat` appelée, tout ajout de texte par la méthode `appendText` conserve la mise en forme en cours.

---

Dans le code suivant, le texte est affecté après l'application de la mise en forme, celle-ci est donc écrasée :

```
var monTexte:TextField = new TextField();  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
addChild ( monTexte );  
  
// création de l'objet TextFormat ( mise en forme )  
var miseEnForme:TextFormat = new TextFormat();  
  
// activation du style gras  
miseEnForme.bold = true;  
  
// modification de la taille du texte  
miseEnForme.size = 14;  
  
// modification de la police  
miseEnForme.font = "Verdana";  
  
// application de la mise en forme  
monTexte.setTextFormat( miseEnForme );  
  
// le contenu est affecté après l'application de la mise en forme  
monTexte.text = "Voici du texte mis en forme";
```

La figure 16-18 illustre le résultat :



*Figure 16-18. Mise en forme du texte inchangée.*

Notons que si nous utilisons la méthode `appendText` après l'application de la mise en forme, celle-ci est conservée. En revanche, si le contenu est remplacé, la mise en forme est perdue. ActionScript 3 intègre une nouvelle propriété `defaultTextFormat` permettant de remédier à cela en définissant un style par défaut :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

// application d'une mise en forme par défaut
monTexte.defaultTextFormat = miseEnForme;

// tout contenu affecté prend la mise en forme par défaut
monTexte.text = "Voici du texte mis en forme";
```

Tout texte ajouté par la suite prendra automatiquement la mise en forme définie par l'objet `miseEnForme`.

Si nous souhaitons modifier le style par défaut, nous pouvons affecter à l'aide de la méthode `setTextFormat` une mise en forme spécifique à l'aide d'un autre objet `TextFormat` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// création de l'objet TextFormat ( mise en forme )
var miseEnForme:TextFormat = new TextFormat();

// activation du style gras
miseEnForme.bold = true;

// modification de la taille du texte
miseEnForme.size = 14;

// modification de la police
miseEnForme.font = "Verdana";

// application d'une mise en forme par défaut
monTexte.defaultTextFormat = miseEnForme;
```

```
// tout contenu affecté prend la mise en forme par défaut
monTexte.text = "Voici du texte mis en forme";

// création de l'objet TextFormat ( mise en forme )
var autreMiseEnForme:TextFormat = new TextFormat();

// modification de la police
autreMiseEnForme.font = "Arial";

// modification de la taille
autreMiseEnForme.size = 18;

// affectation partielle de la nouvelle mise en forme
monTexte.setTextFormat( autreMiseEnForme, 22, 27 );
```

La figure 16-19 illustre le résultat :



*Figure 16-19. Mise en forme du texte partielle.*

La classe `TextField` définit une autre méthode très pratique en matière de mise en forme du texte nommée `getTextFormat` dont voici la signature :

```
public function getTextFormat(beginIndex:int = -1, endIndex:int = -1):TextFormat
```

Analysons chacun des paramètres :

- `beginIndex` : position du caractère de début à partir duquel la mise en forme doit être extraite, la valeur par défaut est -1.
- `endIndex` : position du caractère de fin à partir duquel la mise en forme doit être extraite, la valeur par défaut est -1.

Cette dernière renvoie un objet `TextFormat` associé à la partie du texte spécifiée. Si aucun paramètre n'est spécifié, l'objet `TextFormat` renvoyé concerne la totalité du champ texte.

Comme nous l'avons vu précédemment, lorsque du contenu est affecté à un champ texte, le lecteur Flash crée un objet `TextFormat` interne contenant les options de mise en forme du texte.

Dans le code suivant, nous récupérons les informations de mise en forme du champ `monTexte` :

```
var monTexte:TextField = new TextField();
monTexte.text = "Voici du texte !";
```

```
monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// extraction de l'objet TextFormat
var miseEnForme:TextFormat = monTexte.getTextFormat();

// affiche : Times New Roman
trace( miseEnForme.font );

// affiche : 12
trace( miseEnForme.size );

// affiche : left
trace( miseEnForme.align );

// affiche : 0
trace( miseEnForme.color );

// affiche : false
trace( miseEnForme.bold );

// affiche : false
trace( miseEnForme.italic );

// affiche : false
trace( miseEnForme.underline );
```

En définissant différentes balises HTML, nous remarquons que l'objet `TextFormat` renvoyé reflète correctement la mise en forme actuelle du champ :

```
var monTexte:TextField = new TextField();

monTexte.htmlText = "<font size='18' color='#990000'><i><b>Voici du texte  
!</b></i></font>";

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

// extraction de l'objet TextFormat
var miseEnForme:TextFormat = monTexte.getTextFormat();

// affiche : Times New Roman
trace( miseEnForme.font );

// affiche : 18
trace( miseEnForme.size );

// affiche : left
trace( miseEnForme.align );

// affiche : 990000
trace( miseEnForme.color.toString(16) );

// affiche : true
trace( miseEnForme.bold );

// affiche : true
trace( miseEnForme.italic );
```

```
// affiche : false
trace( miseEnForme.underline );
```

La classe `TextFormat` s'avère extrêmement pratique, nous allons l'utiliser dans la section suivante afin d'augmenter les capacités de la classe `TextField`.

## Etendre la classe `TextField`

Nous avons vu au cours du chapitre 9 intitulé *Etendre les classes natives* qu'il était possible d'augmenter les capacités d'une classe lorsque celle-ci nous paraissait limitée.

La classe `TextField` figure parmi les classes souvent étendues afin d'optimiser l'affichage ou la gestion du texte.

Dans le cas d'applications localisées, la taille du texte peut varier selon les langues utilisées. La taille du champ doit rester la même afin de ne pas perturber la présentation graphique de l'application.

En étendant la classe `TextField` nous allons ajouter un mécanisme d'adaptation automatique du texte au sein du champ. En activant le mécanisme d'adaptation le texte est réduit afin d'être affiché totalement.

La figure 16-20 illustre le résultat :

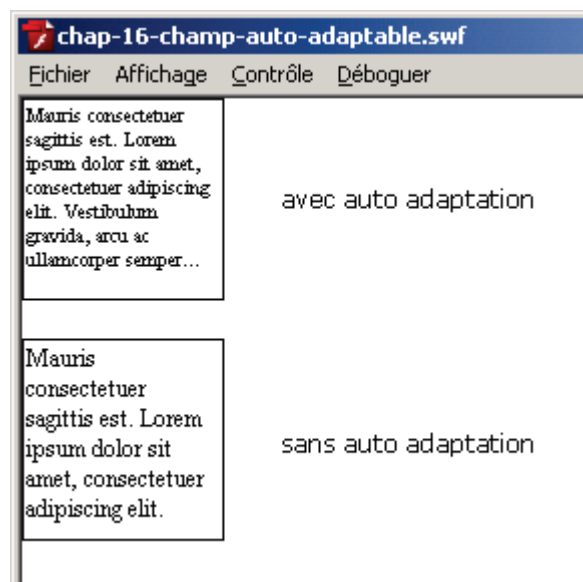


Figure 16-20. Champ texte auto-adaptable.

Dans un nouveau paquetage `org.bytearray.texte` nous définissons la classe `ChampTexteAutoAdaptable` contenant le code suivant :

```
package org.bytearray.texte

{

    import flash.text.TextField;

    public class ChampTexteAutoAdaptable extends TextField

    {

        public function ChampTexteAutoAdaptable ()

        {

        }

    }

}
```

La classe `ChampTexteAutoAdaptable` étend la classe `TextField` afin d’hériter de toutes ses fonctionnalités.

Nous définissons une propriété `autoAdaptation` en lecture écriture afin de spécifier si le champ doit redimensionner ou non le contenu texte :

```
package org.bytearray.texte

{

    import flash.text.TextField;

    public class ChampTexteAutoAdaptable extends TextField

    {

        private var adaptation:Boolean;

        public function ChampTexteAutoAdaptable ()

        {

        }

        public function set autoAdaptation ( pAdaptation:Boolean ):void

        {

            adaptation = pAdaptation;

        }

        public function get autoAdaptation ():Boolean

        {

            return adaptation;

        }

    }

}
```

```
| }
```

Lorsque le contenu texte est ajouté, nous devons déclencher le processus d'adaptation du texte au sein du champ, pour cela nous devons modifier le fonctionnement de la propriété `text`.

Nous surchargeons celle-ci en déclenchant la méthode `adapte` lorsque du contenu est affecté :

```
package org.bytearray.texte

{

    import flash.text.TextField;
    import flash.text.TextFormat;

    public class ChampTexteAutoAdaptable extends TextField

    {

        private var adaptation:Boolean;
        private var formatage:TextFormat;
        private var tailleInitiale:int;

        public function ChampTexteAutoAdaptable ()

        {

        }

        public function set autoAdaptation ( pAdaptation:Boolean ):void

        {

            adaptation = pAdaptation;

        }

        public function get autoAdaptation ():Boolean

        {

            return adaptation;

        }

        public override function set text ( pText:String ):void

        {

            super.text = pText;

            if ( autoAdaptation ) adapte();

        }

        private function adapte ():void

        {

            formatage = getTextFormat();

        }

    }

}
```



```
        tailleInitiale = int(formatage.size);

        while ( textWidth > width || textHeight > height )
        {

            if ( formatage.size <= 0 ) return;

            formatage.size = --tailleInitiale;

            setTextFormat ( formatage );

        }

    }

}
```

Nous retrouvons ici le concept de surcharge en étendant le fonctionnement de la propriété `text`. Bien que celle-ci soit surchargée, nous déclenchons la définition héritée grâce au mot-clé `super`.

La méthode `adapte` trouve une taille de police adéquate afin d'adapter le contenu texte au sein du champ.

Grâce à la propriété `autoAdaptation` nous pouvons activer ou non l'adaptation du texte aux dimensions du champ :

```
import org.bytearray.texte.ChampTexteAutoAdaptable;

var monPremierChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monPremierChamp.border = true;

monPremierChamp.wordWrap = true;

// activation du mode d'auto adaptation
monPremierChamp.autoAdaptation = true;

monPremierChamp.text = "Mauris consectetur sagittis est. Lorem ipsum dolor
sit amet, consectetur adipiscing elit. Vestibulum gravida, arcu ac
ullamcorper semper...";

addChild ( monPremierChamp );

var monSecondChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monSecondChamp.border = true;

monSecondChamp.wordWrap = true;

monSecondChamp.text = "Mauris consectetur sagittis est. Lorem ipsum dolor
sit amet, consectetur adipiscing elit. Vestibulum gravida, arcu ac
ullamcorper semper...";

monSecondChamp.y = 120;

addChild ( monSecondChamp );
```

Dans le code suivant, nous créons un simple champ texte contenant la chaîne de caractères suivante "Bienvenue au camping des Bois Fleuris" :

```
import org.bytearray.texte.ChampTexteAutoAdaptable;

var monPremierChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monPremierChamp.border = true;

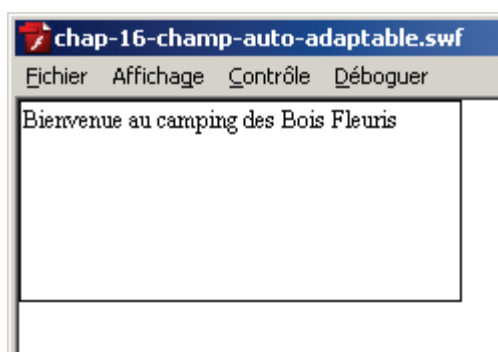
monPremierChamp.width = 220;

monPremierChamp.autoAdaptation = true;

monPremierChamp.text = "Bienvenue au camping des Bois Fleuris";

addChild ( monPremierChamp );
```

La figure 16-21 illustre le rendu :



*Figure 16-21. Contenu texte.*

Le mode d'adaptation du contenu est activé, si nous modifions le contenu du champ par un contenu trop long, le champ réduit automatiquement la taille de la police afin d'afficher totalement le contenu.

Dans le code suivant, le message de bienvenue est localisé en Allemand :

```
import org.bytearray.texte.ChampTexteAutoAdaptable;

var monPremierChamp:ChampTexteAutoAdaptable = new ChampTexteAutoAdaptable();

monPremierChamp.border = true;

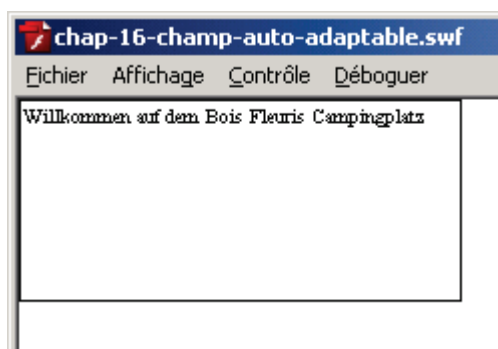
monPremierChamp.width = 220;

monPremierChamp.autoAdaptation = true;

monPremierChamp.text = "Willkommen auf dem Bois Fleuris Campingplatz";

addChild ( monPremierChamp );
```

La figure 16-22 illustre le résultat :



*Figure 16-22. Reduction du texte automatique.*

Dans certains cas, les champs texte peuvent déjà être créés depuis l'environnement auteur afin d'être positionnés convenablement au sein d'une interface.

Au lieu de remplacer chaque champ par une instance de la classe `ChampTexteAutoAdaptable`, nous pouvons directement ajouter le mécanisme d'auto adaptation au sein du prototype de la classe `TextField` :

```
TextField.prototype.texteAutoAdapte = function ( pTexte:String )
{
    this.text = pTexte;
    this.adapte();
}

TextField.prototype.adapte = function ()
{
    var formatage = this.getTextFormat();
    var tailleInitiale = int(formatage.size);

    while ( this.textWidth > this.width || this.textHeight > this.height )
    {
        if ( formatage.size <= 0 ) return;
        formatage.size = --tailleInitiale;
        this.setTextFormat ( formatage );
    }
}
```

Il nous suffit par la suite d'affecter le contenu texte aux différents champs par la méthode `texteAutoAdapte` :

```
champTexte.texteAutoAdapte ( "Willkommen auf dem Bois Fleuris Campingplatz" );
```

Si nous tentons de compiler, le compilateur nous affiche le message suivant :

```
1061: Appel à la méthode texteAutoAdapte peut-être non définie, via la référence de type static flash.text:TextField.
```

Bien entendu, la classe `TextField` ne dispose pas par défaut d'une méthode `texteAutoAdapte`, le compilateur nous empêche donc de compiler.

Nous passons outre cette vérification en transtypant notre champ texte vers la classe dynamique `Object` :

```
Object(champTexte).texteAutoAdapte ( "Willkommen auf dem Bois Fleuris Campingplatz" );
```

A l'exécution, la méthode `texteAutoAdapte` est trouvée et exécutée, le contenu du champ est adapté.

Nous verrons au cours de la section intitulée *Créer un éditeur de texte* un autre exemple d'application des méthodes `setTextFormat` et `getTextFormat`.

## A retenir

- La classe `TextField` peut être étendue afin d'augmenter ses capacités.
- Selon les dimensions du champ, la classe `ChampTexteAutoAdaptable` permet d'adapter la taille de la police afin d'afficher totalement le contenu du texte.

## La classe StyleSheet

Afin de mettre en forme du texte, il peut être intéressant d'externaliser la mise en forme du texte au sein d'une *feuille de style* CSS.

L'avantage de cette technique permet entre autre de mettre à jour la mise en forme du texte sans avoir à recompiler l'application.

La feuille de style est modifiée puis chargée dynamiquement afin de mettre en forme le contenu.

---

Consultez l'aide de Flash CS3 pour la liste des propriétés CSS gérées par le lecteur Flash.

---

Au sein d'un fichier nommé `style.css` nous définissons la feuille de style suivante :

```
| p
```

```

{
    font-family:Verdana, Arial, Helvetica, sans-serif;
    color:#CC9933;
}

.main
{
    font-style:italic;
    font-size:12;
    color:#CC00CC;
}

.title
{
    font-weight:bold;
    font-style:italic;
    font-size:16px;
    color:#FF6633;
}

```

Puis nous chargeons celle-ci à l'aide de l'objet **URLLoader** :

```

var monTexte:TextField = new TextField();

monTexte.wordWrap = true;

monTexte.width = 300;

monTexte.autoSize = TextFieldAutoSize.LEFT;

addChild ( monTexte );

var chargeurCSS:URLLoader = new URLLoader();

// la feuille de style est chargée comme du contenu texte
chargeurCSS.dataFormat = URLLoaderDataFormat.TEXT;

var requete:URLRequest = new URLRequest ( "style.css" );

chargeurCSS.load ( requete );

chargeurCSS.addEventListener (Event.COMPLETE, chargementTermine);
chargeurCSS.addEventListener (IOErrorEvent.IO_ERROR, erreurChargement);

function chargementTermine ( pEvt:Event ):void
{
    // affiche : ( contenu texte de style.css )
    trace( pEvt.target.data );
}

function erreurChargement ( pEvt:IOErrorEvent ):void

```

```
{
    trace( "erreur de chargement" );
}
```

Nous l'affectons au champ texte par la propriété `styleSheet` définie par la classe `TextField` :

```
function chargementTermine ( pEvt:Event ):void
{
    // une feuille de style vide est créée
    var feuilleDeStyle:StyleSheet = new StyleSheet();

    // la feuille de style est définie en interprétant le contenu de style.css
    feuilleDeStyle.parseCSS ( pEvt.target.data );

    // la feuille de style est affectée au champ
    monTexte.styleSheet = feuilleDeStyle;

    // le contenu HTML contenant les balises nécessaires est affecté
    monTexte.htmlText = "<p><span class='main'>Lorem ipsum</span> dolor sit
    amet, consectetur adipiscing elit. <span class='title'>Nulla sit amet
    tellus</span>. Quisque lobortis. Duis tincidunt mollis massa. Maecenas neque
    orci, vulputate eget, consectetur vitae, consectetur ut, urna. Curabitur
    non nibh eu massa tincidunt consequat. Nullam nulla magna, auctor sed, semper
    ut, nonummy a, ipsum. Aliquam pulvinar est sit amet tortor. Sed facilisis
    ligula at est volutpat tristique. Etiam sem felis, facilisis in, fermentum
    at, consectetur quis, ipsum. Morbi tincidunt velit. Integer cursus pretium
    nisl. Fusce et ante.</p>";
}
```

La méthode statique `parseCSS` permet d'interpréter la chaîne de caractère contenue dans le fichier `style.css` afin de définir l'objet `StyleSheet`.

La figure 16-23 illustre la mise en forme :



*Figure 16-23. Mise en forme par feuille de style.*

Grace aux feuilles de style, nous pouvons définir des options de mise en forme liées à l'interactivité.

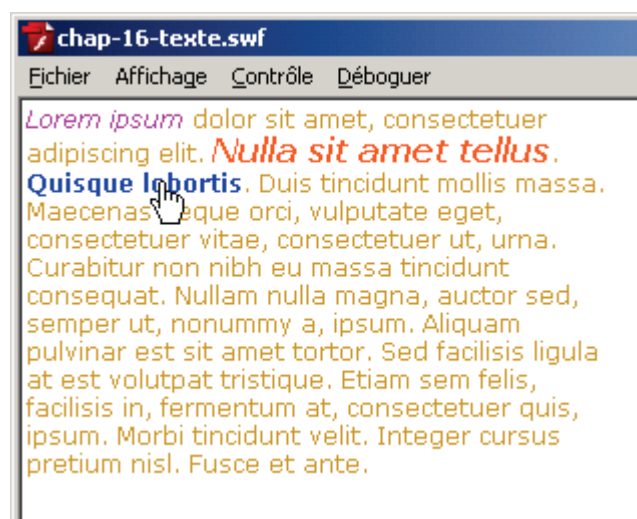
En ajoutant la propriété `a:hover` au sein de la feuille de style nous rendons le texte gras et modifions sa couleur lors du survol :

```
a:hover
{
    font-weight:bold;
    color:#0033CC;
}
```

En ajoutant un lien hypertexte, le texte change de style automatiquement :

```
monTexte.htmlText = "<p><span class='main'>Lorem ipsum</span> dolor sit amet,
consectetuer adipiscing elit. <span class='title'>Nulla sit amet
tellus</span>.<a href='http://www.oreilly.fr'>Quisque lobortis</a>. Duis
tincidunt mollis massa. Maecenas neque orci, vulputate eget, consectetuer
vitae, consectetuer ut, urna. Curabitur non nibh eu massa tincidunt
consequat. Nullam nulla magna, auctor sed, semper ut, nonummy a, ipsum.
Aliquam pulvinar est sit amet tortor. Sed facilisis ligula at est volutpat
tristique. Etiam sem felis, facilisis in, fermentum at, consectetuer quis,
ipsum. Morbi tincidunt velit. Integer cursus pretium nisl. Fusce et
ante.</p>";
```

La figure 16-24 illustre le résultat :

*Figure 16-24. Mise en forme par feuille de style.*

Malheureusement, l'utilisation des CSS s'avère limitée dans certains cas. Un champ texte de saisie n'autorise pas la saisie de texte lorsque celui-ci est lié à une feuille de style.

De la même manière, un champ texte lié à une feuille de style n'est pas modifiable. L'utilisation des méthodes `replaceText`, `replaceSelectedText`, `appendText`, `setTextFormat` n'est pas autorisée.

### A retenir

- Trois techniques permettent de mettre du texte en forme : les balises HTML, l'objet `TextFormat`, la définition d'une feuille style à l'aide de la classe `StyleSheet`.
- Quelle que soit la technique employée, le lecteur Flash fonctionne en interne par l'intermédiaire de balises HTML.

## Modifier le contenu d'un champ texte

Durant l'exécution d'une application, nous avons souvent besoin de modifier le contenu de champs texte. Pour cela, nous pouvons utiliser les propriétés suivantes :

- `text` : permet l'affectation de contenu non HTML ;
- `htmlText` : permet l'affectation de contenu HTML ;

Ainsi que les méthodes suivantes :

- `appendText` : remplace l'opérateur `+=` lors d'ajout de texte. Ne prend pas en charge le texte au format HTML ;
- `replaceText` : permet de remplacer une partie du texte au sein d'un champ. Ne prend pas en charge le texte au format HTML.

Dans le cas d'ajout de contenu nous pouvons utiliser l'opérateur `+=` sur la propriété `text` :

```
var monTexte:TextField = new TextField();
addChild ( monTexte );
monTexte.autoSize = TextFieldAutoSize.LEFT;
monTexte.text = "ActionScript";
monTexte.text += " 3"
// affiche : ActionScript 3
trace( monTexte.text );
```

Cependant, en ActionScript 3 l'utilisation de l'opérateur `+=` sur un champ texte est dépréciée au profit de la méthode `appendText`.

Si le mode *Avertissements* du compilateur est activé, celui-ci nous avertit de la dépréciation de l'opérateur `+=` :



```
Warning: 3551: L'ajout de texte à la fin d'un champ texte TextField avec +=  
est beaucoup plus lent que l'utilisation de la méthode  
TextField.appendText().
```

Voici la signature de la méthode `appendText` :

```
public function appendText(newText:String):void
```

Nous passons en paramètre le texte à ajouter au champ de la manière suivante :

```
var monTexte:TextField = new TextField();  
  
addChild ( monTexte );  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
monTexte.text = "ActionScript";  
  
monTexte.appendText ( " 3" );  
  
// affiche : ActionScript 3  
trace( monTexte.text );
```

En faisant un simple test de performances, nous voyons que l'opérateur `+=` est en effet plus lent que la méthode `appendText` :

```
var monTexte:TextField = new TextField();  
  
addChild ( monTexte );  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
var debut:Number = getTimer();  
  
for (var i:int = 0; i< 1500; i++ )  
  
{  
  
    monTexte.text += "ActionScript 3";  
  
}  
  
// affiche : 1120  
trace( getTimer() - debut );
```

Une simple concaténation sur 1500 itérations nécessite 1120 millisecondes.

Si nous utilisons cette fois la méthode `appendText` :

```
var monTexte:TextField = new TextField();  
  
addChild ( monTexte );  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
var debut:Number = getTimer();  
  
for (var i:int = 0; i< 1500; i++ )
```

```
{
    monTexte.appendText ( "ActionScript 3" );
}

// affiche : 847
trace( getTimer() - debut );
```

Le temps d'exécution est de 847 ms. L'utilisation de la méthode `appendText` s'avère être environ 30% plus rapide que l'opérateur `+=`.

Nous pourrions penser que la méthode `appendText` s'avère donc être la méthode favorite. En réalité, afin d'affecter du texte de manière optimisée, il convient de limiter au maximum la manipulation du champ texte.

Ainsi, nous pourrions accélérer le code précédent en créant une variable contenant le texte, puis en l'affectant au champ une fois la concaténation achevée :

```
var monTexte:TextField = new TextField();
addChild ( monTexte );
monTexte.autoSize = TextFieldAutoSize.LEFT;
var debut:Number = getTimer();
var contenu:String = monTexte.text;
for (var i:int = 0; i< 1500; i++ )
{
    contenu += "ActionScript 3";
}
monTexte.text = contenu;

// affiche : 2
trace( getTimer() - debut );
```

En procédant ainsi, nous passons à une vitesse d'exécution de 2 millisecondes. Soit un temps d'exécution environ 420 fois plus rapide que la méthode `appendText` et 560 fois plus rapide que l'opérateur `+=` auprès de la propriété `text`.

La même approche peut être utilisée dans le cas de manipulation de texte au format HTML grâce à la propriété `htmlText` :

```
var monTexte:TextField = new TextField();
addChild ( monTexte );
monTexte.autoSize = TextFieldAutoSize.LEFT;
```

```
var debut:Number = getTimer();

var contenu:String = monTexte.htmlText;

for (var i:int = 0; i< 1500; i++ )
{
    contenu += "<b>ActionScript<b> 3";
}

monTexte.htmlText = contenu;

// affiche : 29
trace( getTimer() - debut );
```

Afin d'obtenir le même résultat, si nous concaténons directement le contenu au sein du champ texte, le temps d'exécution excède 15 secondes et lève une exception de type `ScriptTimeoutError` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

var debut:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    monTexte.htmlText += "ActionScript <b>2</b>";
}

trace( getTimer() - debut );
```

Le processus de rendu est trop complexe et lève l'exception suivante :

```
Error: Error #1502: La durée d'exécution d'un script excède le délai par
défaut (15 secondes).
```

Il convient donc d'utiliser la méthode `appendText` ou les propriétés `text` et `htmlText` dans un contexte non intensif.

Dans tous les autres cas, nous préférons créer une variable contenant le texte, puis travailler sur celle-ci avant de l'affecter au champ texte.

## Remplacer du texte

Si nous souhaitons modifier une partie du texte, nous pouvons utiliser la méthode `replaceText` de la classe `TextField` ayant la signature suivante :

```
public function replaceText(beginIndex:int, endIndex:int,
newText:String):void
```

Cette méthode accepte trois paramètres, dont voici le détail :

- `beginIndex` : la position du caractère à partir duquel le remplacement démarre, la valeur par défaut est -1 ;
- `endIndex` : la position du caractère à partir duquel le remplacement se termine, la valeur par défaut est -1 ;
- `newText` : le texte de remplacement ;

Afin de bien comprendre l'utilisation de cette méthode, prenons le cas du texte suivant :

```
| ActionScript 2
```

Ce texte est ajouté à un champ texte :

```
| var monTexte:TextField = new TextField();  
addChild ( monTexte );  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
monTexte.text = "ActionScript 2";
```

Afin de remplacer le chiffre 2 par 3 nous utilisons la méthode `replaceText` :

```
| var monTexte:TextField = new TextField();  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
monTexte.text = "ActionScript 2";  
monTexte.replaceText( 13, 14, "3" );  
addChild ( monTexte );
```

Le champ texte contient alors le texte suivant :

```
| ActionScript 3
```

Nous pouvons rendre le remplacement dynamique et rechercher la position du caractère à l'aide de la méthode `indexOf` de la classe `String` :

```
| var monTexte:TextField = new TextField();  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
monTexte.text = "ActionScript 2";  
var chaine:String = "2";  
var position:int = monTexte.text.indexOf ( chaine );  
if ( position != -1 ) monTexte.replaceText ( position,  
position+chaine.length, "3" );  
addChild ( monTexte );
```

Comme nous l'avons vu lors du chapitre 2 intitulé *Langage et API du lecteur Flash*, l'intégration des expressions régulières en ActionScript 3 rend la manipulation de chaînes de caractères extrêmement puissante.

Il est préférable d'utiliser les méthodes de la classe `String` telles `search`, `replace` ou `match` pour une recherche plus complexe.

### A retenir

- Afin de modifier le contenu d'un champ texte, nous pouvons utiliser la propriété `text`, `htmlText` ainsi que les méthodes `appendText` ou `replaceText`.
- La méthode `appendText` ne permet pas l'affectation de contenu HTML.
- Dans tous les cas, l'utilisation d'une variable temporaire afin de concaténer et traiter le contenu s'avère plus rapide.
- La méthode `replaceText` permet de remplacer du texte au sein d'un champ de manière simplifiée.

## L'événement `TextEvent.LINK`

Les précédentes versions d'ActionScript intégraient une fonctionnalité appelée `asfunction` permettant de déclencher des fonctions ActionScript à partir de texte HTML. Ce protocole fut intégré au sein du lecteur Flash 5.

Afin de déclencher une fonction ActionScript, le nom de la fonction précédé du mot-clé `asfunction` était passé à l'attribut `href` de la balise d'ancrage `<a>` :

```
var monTexte:TextField = this.createTextField ("legende", 0, 0, 0, 0, 0);

monTexte.autoSize = true;

function maFonction ( pArgument )
{
    trace ( "Le paramètre passé est " + pArgument );
}

monTexte.html = true;

monTexte.htmlText = "<a href='asfunction:maFonction,Coucou'>Cliquez ici  
!</a>";
```

Afin de rester conforme au nouveau modèle événementiel, ActionScript 3 remplace ce protocole en proposant la diffusion d'événements `TextEvent.LINK` par les champs texte.

Dans le code suivant, nous créons un champ texte comportant une balise `<b>` et `<i>` :

```
var monTexte:TextField = new TextField();

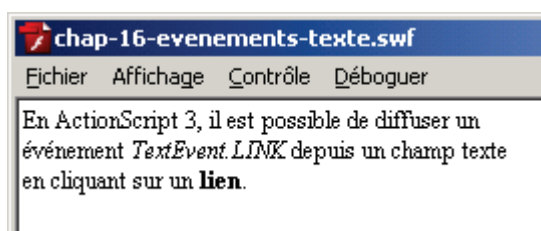
monTexte.width = 250;
monTexte.wordWrap = true;

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "En ActionScript 3, il est possible de diffuser un
événement <i>TextEvent.LINK</i> depuis un champ texte en cliquant sur un
<b>lien</b>.";
```

La figure 16-25 illustre le résultat :



*Figure 16-25. Champ texte formaté.*

Nous allons permettre la diffusion d'un événement `TextEvent.LINK`, à l'aide de la syntaxe suivante :

```
<a href='event:paramètre'>Texte Cliquable</a>
```

L'utilisation du mot clé `event` en tant que valeur de l'attribut `href` permet la diffusion d'un événement `TextEvent.LINK` lors du clic sur le lien. La valeur précisée après les deux points est affectée à la propriété `text` de l'objet événementiel diffusé.

Dans le code suivant, nous écoutons l'événement `TextEvent.LINK` auprès du champ texte :

```
var monTexte:TextField = new TextField();

monTexte.width = 250;
monTexte.wordWrap = true;

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "En ActionScript 3, il est possible de diffuser un
événement <i>TextEvent.LINK</i> depuis un champ texte en cliquant sur un <a
href='event:Texte caché !'><b>lien</b></a>.";

monTexte.addEventListener (TextEvent.LINK, clicLien);

function clicLien ( pEvt:TextEvent ):void
```

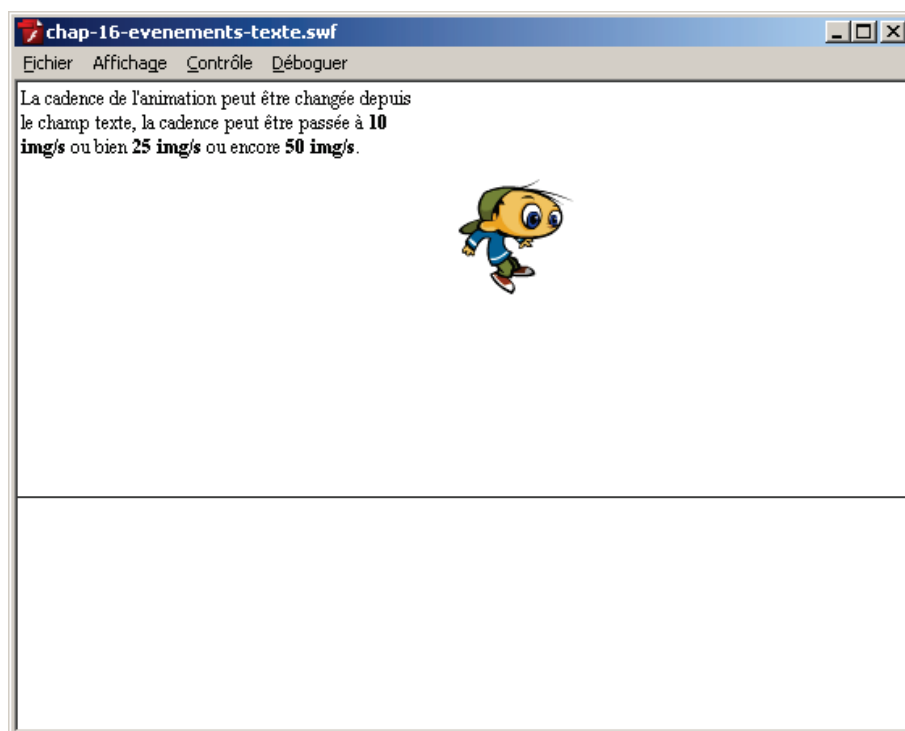
```
{  
    // affiche : [TextEvent type="link" bubbles=true cancelable=false  
    eventPhase=2 text="Texte caché !"]  
    trace( pEvt );  
  
    // affiche : [object TextField]  
    trace( pEvt.target );  
}
```

La propriété `text` de l'objet événementiel contient le texte spécifié après l'attribut `event`.

Dans le code suivant, nous changeons dynamiquement la cadence de l'animation en cliquant sur différents liens associés au champ texte :

```
var monTexte:TextField = new TextField();  
  
monTexte.width = 250;  
monTexte.wordWrap = true;  
  
addChild ( monTexte );  
  
monTexte.autoSize = TextFieldAutoSize.LEFT;  
  
monTexte.htmlText = "La cadence de l'animation peut être changée depuis le  
champ texte, la cadence peut être passée à <a href='event:10'><b>10  
img/s</b></a> ou bien <a href='event:25'><b>25 img/s</b></a> ou encore <a  
href='event:50'><b>50 img/s</b></a>.";  
  
monTexte.addEventListener (TextEvent.CLICK, onLinkClick);  
  
function onLinkClick ( pEvt:TextEvent ):void  
{  
    // affiche : 25  
    trace( pEvt.text );  
  
    stage.frameRate = int ( pEvt.text );  
}
```

La figure 16-26 illustre le résultat :



*Figure 16-26. Modification dynamique de la cadence de l'animation depuis le champ texte.*

Il n'est pas possible de diffuser d'autres événements à l'aide de l'attribut `event` de la balise `<a>`. De la même manière, il n'est pas prévu de pouvoir passer plusieurs paramètres à la fonction écouteur, mais à l'aide d'un séparateur comme une virgule nous pouvons y parvenir :

```
| <a href='event:paramètre1,paramètre2,paramètre3'>Texte Cliquable</a>
```

Ainsi, nous pouvons passer plusieurs cadences avec la syntaxe suivante :

```
| <a href='event:10,20,30,40'><b>10 img/s</b></a>
```

Au sein de la fonction écouteur nous transformons la chaîne en un tableau à l'aide de la méthode `split` de la classe `String` :

```
function clicLien ( pEvt:TextEvent ):void
{
    var parametres:Array = pEvt.text.split (",");

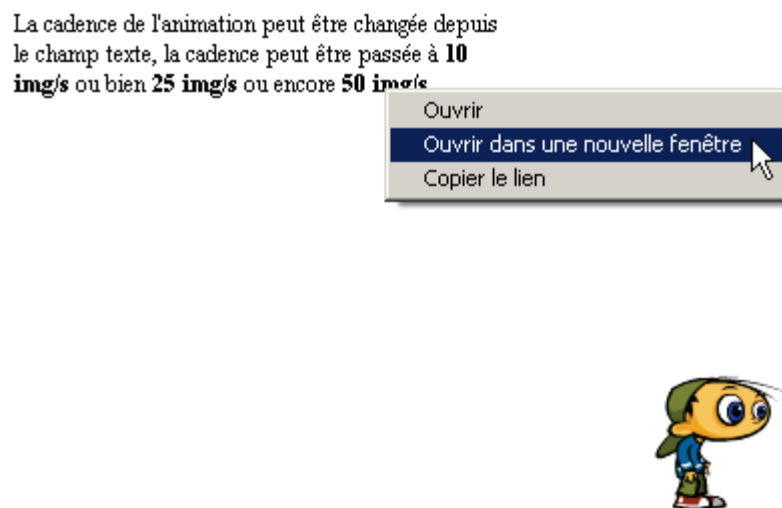
    /* affiche :
    10
    20
    30
    40
    */
    for each ( var p:* in parametres ) trace( p );
}
```



}

L'utilisation des événements `TextEvent.LINK` entraîne malheureusement un comportement pénalisant.

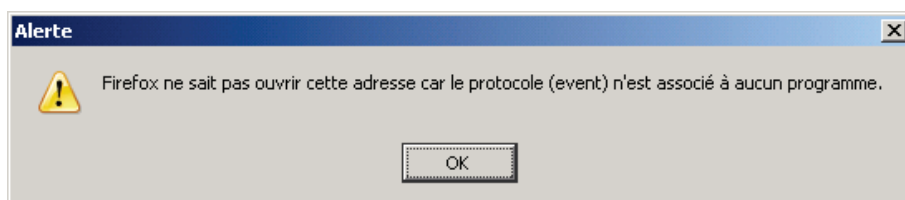
Lorsqu'un clic droit est détecté sur un lien associé au champ texte, un menu contextuel s'ouvre habituellement comme l'illustre la figure 16-27 :



*Figure 16-27. Menu contextuel de liens hypertexte.*

L'utilisation d'événements `TextEvent.LINK` empêche le bon fonctionnement du menu contextuel.

Si nous décidons d'ouvrir le lien directement ou dans une nouvelle fenêtre, le message d'erreur illustré en figure 16-28 s'affiche :



*Figure 16-28. Message d'erreur associé aux liens hypertexte.*

Le navigateur par défaut ne reconnaît pas le lien associé (protocole event) qui est représenté sous la forme suivante :

`event:paramètre`

De plus en sélectionnant une option du menu contextuel l'événement `TextEvent.LINK` n'est pas diffusé.

Il convient donc de prendre ces comportements en considération.

## A retenir

- Le protocole `asfunction` est remplacé par la diffusion d'événements `TextEvent.LINK`.
- Afin de diffuser un événement `TextEvent.LINK`, nous utilisons le mot-clé `event` en tant que valeur de l'attribut `href` de la balise d'ancrage `<a>`.
- L'utilisation d'événements `TextEvent.LINK` empêche le bon fonctionnement du menu contextuel associé au lien.

## Charger du contenu externe

Nous avons vu qu'il était possible d'intégrer des balises HTML au sein d'un champ texte. Dans le cas de l'utilisation de la balise `<img>`, nous pouvons intégrer facilement un élément multimédia telle une image ou un SWF.

Dans le code suivant, nous intégrons une image au champ texte :

```
var monTexte:TextField = new TextField();
addChild ( monTexte );
monTexte.autoSize = TextFieldAutoSize.LEFT;
monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'>";
```

La figure 16-29 illustre le résultat :



Figure 16-29. Image intégrée dans un champ texte.

Afin de correctement gérer le chargement des médias intégrés aux champs texte, ActionScript 3 introduit une nouvelle méthode de la

classe `TextField` nommée `getImageReference` dont voici la signature :

```
public function getImageReference(id:String):DisplayObject
```

Lorsqu'une balise `<img>` est utilisée au sein d'un champ texte, le lecteur Flash crée automatiquement un objet `Loader` enfant au champ texte associé au média chargé. Bien que la classe `TextField` ne soit pas un objet de type `DisplayObjectContainer`, l'objet `Loader` est pourtant contenu par le champ texte.

Dans le code suivant, nous ajoutons un identifiant `monLogo` à l'image intégrée grâce à l'attribut `id` :

```
monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte  
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'  
id='monLogo'>";
```

Afin d'extraire l'objet `Loader` associé au média chargé, nous passons ce même identifiant à la méthode `getImageReference` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte  
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'  
id='monLogo'>";

var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monLogo") );

// affiche : [object Loader]
trace( chargeurLogo );

// affiche : [object TextField]
trace( chargeurLogo.parent );
```

Nous remarquons que la propriété `parent` de l'objet `Loader` fait bien référence au champ texte.

Nous pouvons ainsi gérer le préchargement de l'image et accéder à celle-ci grâce la propriété `content` de l'objet `Loader` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte  
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'  
id='monLogo'>";

var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monLogo") );

// écoute de l'événement Event.COMPLETE
```

```

chargeurLogo.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

function chargementTermine ( pEvt:Event ):void
{
    // affiche : [object Bitmap]
    trace( pEvt.target.content );
}

```

Dans le code suivant, nous affectons un filtre de flou à l'image chargée :

```

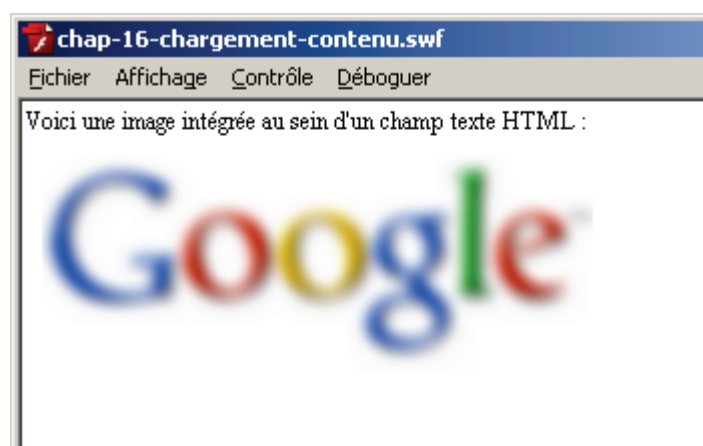
var point:Point = new Point ( 0, 0 );
var filtreFlou:BlurFilter = new BlurFilter ( 8, 8, 2 );

function chargementTermine ( pEvt:Event ):void
{
    if ( pEvt.target.content is Bitmap )
    {
        var imageChargee:Bitmap = Bitmap ( pEvt.target.content );

        // affecte un filtre de flou à l'image intégrée au champ texte
        imageChargee.bitmapData.applyFilter ( imageChargee.bitmapData,
        imageChargee.bitmapData.rect, point, filtreFlou );
    }
}

```

La figure 16-30 illustre l'image floutée :



*Figure 16-30. Image chargée dynamiquement puis floutée.*

Une fois l'image chargée, celle-ci est entièrement manipulable.

Si aucune image n'est associée à l'identifiant passé à la méthode `getImageReference`, celle-ci renvoie `null`.

Ainsi, le chargement d'une image au sein d'un champ texte repose sur les mêmes mécanismes de chargement qu'une image traditionnelle chargée directement au sein d'un objet `Loader` :

```
var monTexte:TextField = new TextField();

addChild ( monTexte );

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.htmlText = "<p>Voici une image intégrée au sein d'un champ texte
HTML :</p><p><img src='http://www.google.com/intl/en_ALL/images/logo.gif'
id='monLogo'>";

var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monLogo") );

// écoute des différents événements
chargeurLogo.contentLoaderInfo.addEventListener ( Event.OPEN,
chargementDemarre );
chargeurLogo.contentLoaderInfo.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
chargeurLogo.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );
chargeurLogo.contentLoaderInfo.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );

function chargementDemarre ( pEvt:Event ):void
{
    trace("chargement démarré");
}

function chargementEnCours ( pEvt:ProgressEvent ):void
{
    trace("chargement en cours : " + pEvt.bytesLoaded + " / " + pEvt.bytesTotal
);
}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}

var point:Point = new Point ( 0, 0 );
var filtreFlou:BlurFilter = new BlurFilter ( 8, 8, 2 );

function chargementTermine ( pEvt:Event ):void
{

```

```
if ( pEvt.target.content is Bitmap )
{
    var imageChargee:Bitmap = Bitmap ( pEvt.target.content );

    // affecte un filtre de flou à l'image intégrée au champ texte
    imageChargee.bitmapData.applyFilter ( imageChargee.bitmapData,
    imageChargee.bitmapData.rect, point, filtreFlou );
}
}
```

Grâce aux différents événements diffusés par l'objet `LoaderInfo`, nous pouvons gérer les erreurs de chargement des images associées au champ texte en remplaçant l'image non trouvée par une image de substitution.

De la même manière, nous pouvons intégrer dans le champ texte un SWF.

```
monTexte.htmlText = "<p>Voici une animation intégrée au sein d'un champ texte  
HTML :</p><p><img src='animation.swf' id='monAnim'>";
```

Grâce à la méthode `getImageReference` nous extrayons l'objet `Loader` associé :

```
var chargeurLogo:Loader = Loader ( monTexte.getImageReference("monAnim") );
```

Une fois le chargement terminé, nous accédons aux informations liées au SWF chargé et modifions son opacité :

```
function chargementTermine ( pEvt:Event ):void
{
    pEvt.target.content.alpha = .5;

    if ( pEvt.target.content is DisplayObjectContainer )
    {
        // affiche : 12
        trace( pEvt.target.frameRate );

        // affiche : application/x-shockwave-flash
        trace( pEvt.target.contentType );

        // affiche : 9
        trace( pEvt.target.swfVersion );

        // affiche : 3
        trace( pEvt.target.actionScriptVersion );
    }
}
```

La figure 16-31 illustre le résultat :



*Figure 16-31. Animation avec opacité réduite.*

Il est important de signaler que seule la méthode `getImageReference` permet d'extraire l'objet `Loader` associé au média chargé.

Bien que l'objet `TextField` contienne les objets `Loader`, la classe `TextField` n'hérite pas de la classe `DisplayObjectContainer` et n'offre malheureusement pas de propriétés telles `numChildren` ou de méthodes `getChildAt` ou `removeChild`.

## A retenir

- Un objet `Loader` est créé pour chaque média intégré à un champ texte.
- Bien que la classe `TextField` n'hérite pas de la classe `DisplayObjectContainer`, les objets `Loader` sont enfants du champ texte.
- Seule la méthode `getImageReference` permet d'extraire l'objet `Loader` associé au média chargé.
- Afin de pouvoir utiliser la méthode `getImageReference` un identifiant doit être affecté au média à l'aide de l'attribut `id`.

## Exporter une police dans l'animation

Lorsque nous créons un champ texte par programmation, le lecteur utilise par défaut les polices du système d'exploitation.

Même si ce mécanisme possède un intérêt évident lié au poids de l'animation, si l'ordinateur visualisant l'application ne possède pas la police nécessaire, le texte risque de ne pas être affiché.

Pour nous rendre compte d'autres limitations liées aux polices systèmes, nous pouvons tester le code suivant :

```
var monTexte:TextField = new TextField()
monTexte.autoSize = TextFieldAutoSize.LEFT;
monTexte.text = "Police système";
monTexte.rotation = 45;
addChild ( monTexte );
```

Nous remarquons que le texte n'est pas affiché, car nous avons fait subir une rotation de 45 degrés au champ texte. Sans les contours de polices intégrés, le lecteur est incapable de travailler sur la rotation, les effets de masque ou l'opacité du texte.

Si nous tentons de modifier l'opacité, le texte demeure totalement opaque :

```
var monTexte:TextField = new TextField();
monTexte.autoSize = TextFieldAutoSize.LEFT;
monTexte.text = "Police système";
monTexte.alpha = .4;
addChild ( monTexte );
```

Afin de remédier à ces limitations nous devons intégrer la police à l'animation.

Pour cela, nous cliquons sur la partie droite de la bibliothèque du document en cours comme l'illustre la figure 16-32 :



*Figure 16-32. Animation avec opacité réduite.*

Puis nous sélectionnons l'option *Nouvelle police*.

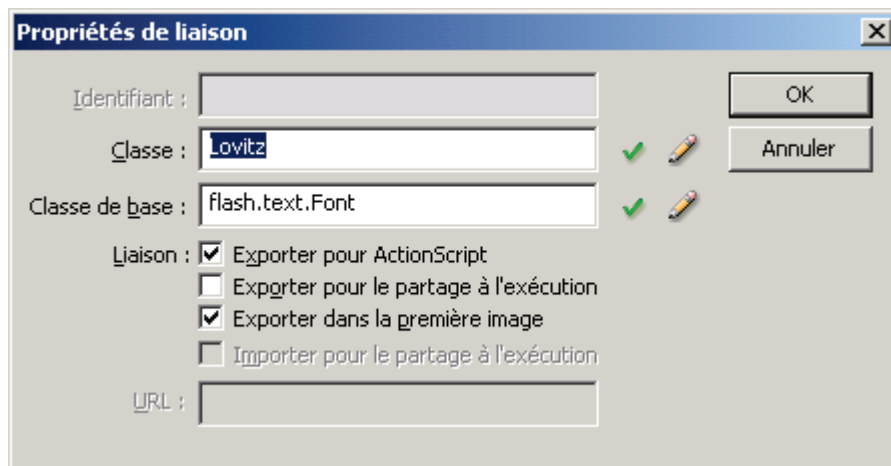


Au sein du nouveau panneau, nous sélectionnons la police à intégrer puis nous validons.

Dans notre cas, nous utilisons la police `Lovitz`.

Notez que le nom de la police spécifié dans le champ associé n'a pas d'incidence sur la suite de l'opération.

En utilisant le panneau *Propriétés de liaison* du symbole de police, nous associons une classe auto générée comme l'illustre la figure 16-33 :



*Figure 16-33. Classe de police auto générée.*

Bien entendu, le nom de la classe associée ne doit pas nécessairement posséder le nom de la police, nous aurions pu utiliser `MaPolice`, `PolicePerso` ou autres.

Afin d'utiliser une police embarquée dans un champ texte, nous utilisons dans un premier temps la propriété `embedFonts` de la classe `TextField` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

// le champ texte doit utiliser une police embarquée
monTexte.embedFonts = true;
```

Puis, nous indiquons la police à utiliser en instanciant la classe `Lovitz`, et en passant son nom à la propriété `font` d'un objet de mise forme `TextFormat` :

```
var monTexte:TextField = new TextField();

monTexte.autoSize = TextFieldAutoSize.LEFT;

// le champ doit utiliser une police embarquée
```

```
monTexte.embedFonts = true;

// nous instancions la police
var policeBibliotheque:Lovitz = new Lovitz();

// un objet de mise en forme est créé
var miseEnForme:TextFormat = new TextFormat();

// nous passons le nom de la police embarquée à l'objet TextFormat
miseEnForme.font = policeBibliotheque.fontName;

miseEnForme.size = 64;

// affectation de la mise en forme
monTexte.defaultTextFormat = miseEnForme;

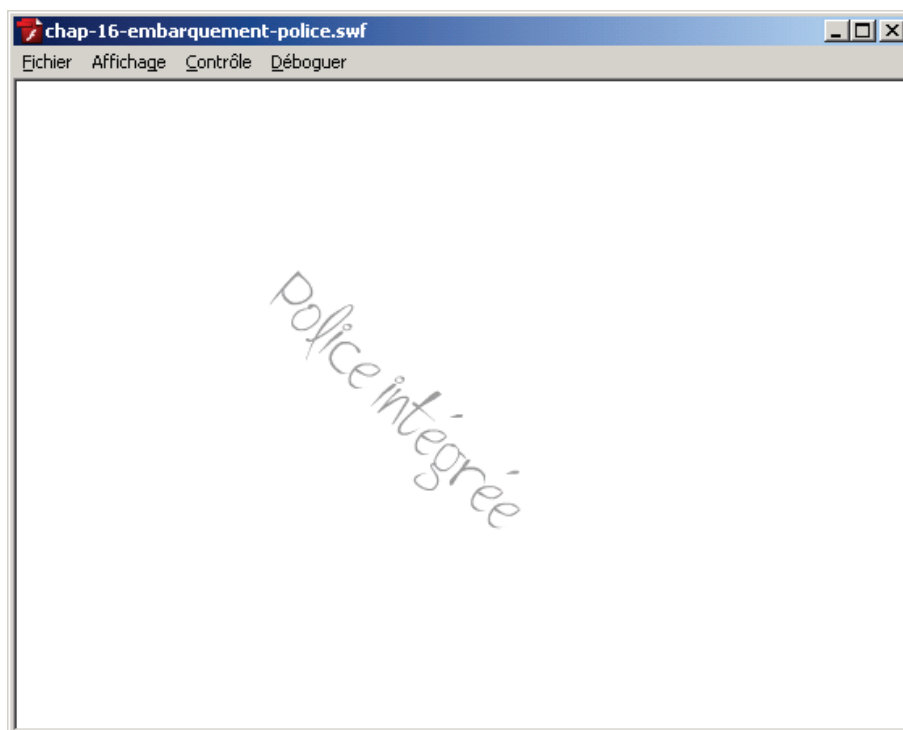
monTexte.text = "Police intégrée";

monTexte.rotation = 45;
monTexte.alpha = .4;

monTexte.x = ( stage.stageWidth - monTexte.width ) / 2;
monTexte.y = ( stage.stageHeight - monTexte.height ) / 2;

addChild ( monTexte );
```

La figure 16-34 illustre le champ texte utilisant la police embarquée :



*Figure 16-34. Champ texte utilisant une police embarquée.*

Malheureusement, en exportant les contours de polices dans l'animation, nous augmentons sensiblement le poids du SWF généré.

Même si cela peut paraître négligeable dans un premier temps, cela peut poser un réel problème dans le cas d'applications localisées, où plusieurs polices peuvent être nécessaires.

Au lieu d'intégrer plusieurs polices au sein de l'animation, nous préférons charger dynamiquement la police nécessaire lorsque l'application en fait la demande.

## Charger dynamiquement une police

Une des limitations des précédentes versions d'ActionScript concernait le manque de souplesse lié au chargement de polices dynamique. ActionScript 3 simplifie grandement ce processus grâce à l'introduction de la classe `flash.text.Font`.

Nous avons découvert au cours du chapitre 13 intitulé *Charger du contenu* la méthode `getDefinition` de la classe `ApplicationDomain`. Au cours de ce chapitre, nous avons extrait différentes définitions de classes issues d'un SWF chargé dynamiquement. Le même concept peut être appliqué dans le cas de polices associées à des classes. En intégrant une police au sein d'un SWF, nous allons pouvoir charger ce dernier puis en extraire la police.

Une fois la police en bibliothèque et associée à une classe, nous exportons l'animation sous la forme d'un SWF appelé `bibliotheque.swf`.

Dans un nouveau document Flash CS3 nous chargeons dynamiquement cette bibliothèque partagée afin d'extraire la définition de classe de police :

```
var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargeurTerminé );

chargeur.load ( new URLRequest("bibliotheque.swf") );

function chargeurTerminé ( pEvt:Event ):void
{
    var DefinitionClasse:Class = Class (
    pEvt.target.applicationDomain.getDefinition("Lovitz") );

    // affiche : [class Lovitz]
    trace( DefinitionClasse );
}
```

Grâce à la méthode statique `registerFont` de la classe `flash.text.Font` nous enregistrons la définition de classe comme nouvelle police disponible au sein de tous les SWF de l'application :

```
function chargementTermine ( pEvt:Event ):void
{
    var DefinitionClasse:Class = Class (
    pEvt.target.applicationDomain.getDefinition("Lovitz") );

    Font.registerFont ( DefinitionClasse );
}
```

La méthode statique `enumerateFonts` de la classe `Font` nous indique que la police a été correctement intégrée :

```
function chargementTermine ( pEvt:Event ):void
{
    var DefinitionClasse:Class = Class (
    pEvt.target.applicationDomain.getDefinition("Lovitz") );

    // énumération des polices intégrées
    var policeIntegrees:Array = Font.enumerateFonts();

    // affiche : 0
    trace( policeIntegrees.length );

    Font.registerFont ( DefinitionClasse );

    policeIntegrees = Font.enumerateFonts();

    // affiche : 1
    trace( policeIntegrees.length );

    // affiche : [object Lovitz]
    trace( policeIntegrees[0] );
}
```

Enfin, nous créons un champ texte et affectons la police chargée dynamiquement à l'aide d'un objet `TextFormat` :

```
var monTexte:TextField = new TextField();
monTexte.embedFonts = true;
monTexte.autoSize = TextFieldAutoSize.LEFT;
addChild ( monTexte );

var chargeur:Loader = new Loader();
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

chargeur.load ( new URLRequest("bibliotheque.swf") );

function chargementTermine ( pEvt:Event ):void
{
```

```

    var DefinitionClasse:Class = Class (
    pEvt.target.applicationDomain.getDefinition("Lovitz") );

    Font.registerFont ( DefinitionClasse );

    var policeBibliotheque:Font = new DefinitionClasse();

    monTexte.defaultTextFormat = new TextFormat(policeBibliotheque.fontName,
    64, 0);

    monTexte.htmlText = "Police chargée dynamiquement !";

}

```

La figure 16-35 illustre le résultat :



*Figure 16-35. Formatage du texte à l'aide d'une police dynamique.*

Une fois la police chargée et enregistrée parmi la liste des polices disponibles, nous pouvons l'utiliser en conjonction avec une feuille de style.

Nous ajoutons le nom de la police officiel grâce à l'attribut `font-family` de la feuille de style :

```

.main
{
    font-family:Lovitz;
    font-style:italic;
    font-size:42;
    color:#CC00CC;
}

```

Attention, le nom de police utilisée ici doit être le nom *officiel* de la police et non le nom de la classe de police associée.

Puis nous associons une feuille de style ayant recours à la police dynamique :

```

var monTexte:TextField = new TextField();

monTexte.embedFonts = true;
monTexte.rotation = 45;

monTexte.autoSize = TextFieldAutoSize.LEFT;

monTexte.wordWrap = true;

```

```
monTexte.width = 250;

addChild ( monTexte );

var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementPoliceTermine );

chargeur.load ( new URLRequest("bibliotheque.swf") );

var chargeurCSS:URLLoader = new URLLoader();

chargeurCSS.dataFormat = URLLoaderDataFormat.TEXT;

chargeurCSS.addEventListener ( Event.COMPLETE, chargementCSSTermine );

function chargementPoliceTermine ( pEvt:Event ):void
{
    var definitionClasse:Class = Class (
pEvt.target.applicationDomain.getDefinition("Lovitz") );

    Font.registerFont ( definitionClasse );

    var requete:URLRequest = new URLRequest ("style.css");

    // une fois la police chargée et enregistrée nous chargeons la feuille de
style
    chargeurCSS.load ( requete );
}

function chargementCSSTermine ( pEvt:Event ):void
{
    var feuilleDeStyle:StyleSheet = new StyleSheet();

    feuilleDeStyle.parseCSS ( pEvt.target.data );

    monTexte.styleSheet = feuilleDeStyle;

    monTexte.htmlText = "<span class='main'>Lorem ipsum dolor sit amet,
consectetuer adipiscing elit. Donec ligula. Proin tristique. Sed semper enim
at augue. In dictum. Pellentesque pellentesque dui pulvinar nisi. Fusce eu
tortor non lorem semper iaculis. Pellentesque nisl dui, lacinia vitae,
vehicula a, pellentesque eget, urna. Aliquam erat volutpat. Praesent et massa
vel augue aliquam iaculis. In et quam. Nulla ut ligula. Ut porttitor.
Vestibulum elit purus, auctor non, commodo sit amet, vestibulum ut,
lorem.</span>";
}

monTexte.x = ( stage.stageWidth - monTexte.width ) / 2;
monTexte.y = ( stage.stageHeight - monTexte.height ) / 2;
```

La figure 16-36 illustre le rendu du texte :

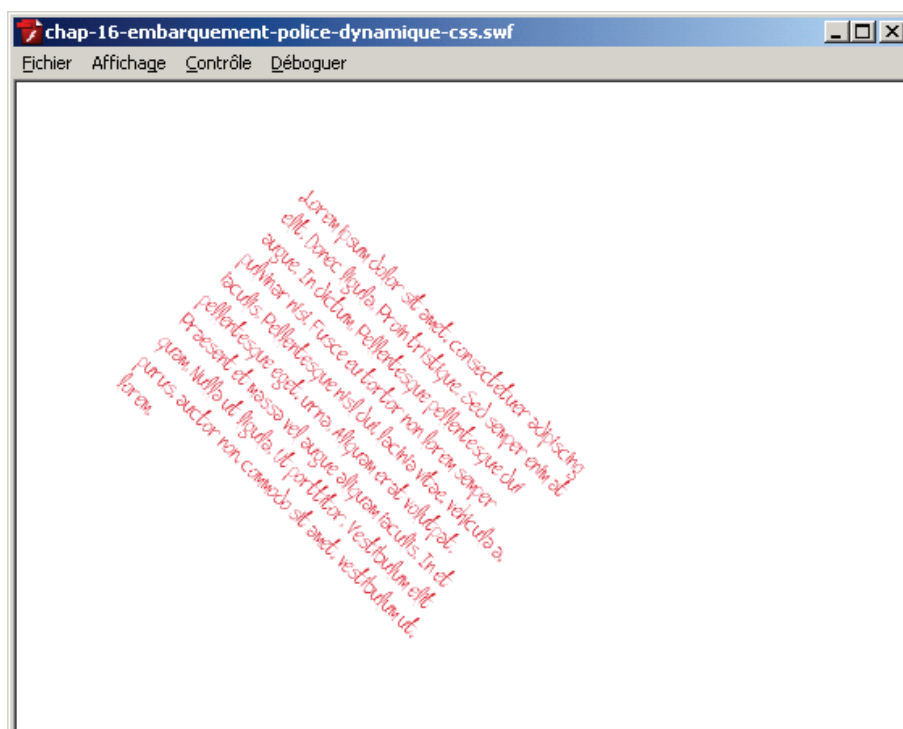


Figure 16-36. Police dynamique et feuille de style.

La rotation du texte nous confirme que la police `Lovitz` est correctement intégrée à l'application, le rendu du texte est donc assuré quelque soit le poste visionnant l'animation.

## A retenir

- ActionScript 3 simplifie le chargement de police dynamique.
- La police est intégrée à un SWF, puis chargée dynamiquement.
- La méthode `getDefinition` de l'objet `ApplicationDomain` permet d'extraire la définition de classe.
- La méthode statique `registerFont` de la classe `Font` permet d'enregistrer la définition de classe dans la liste globale des polices disponibles. La police est alors accessible auprès de tous les SWF de l'application.

## Détecter les coordonnées de texte

Nous avons vu que de nombreuses méthodes ont été ajoutées à la classe `TextField` en ActionScript 3. Nous allons tirer profit d'une nouvelle méthode appelée `getCharBoundaries` dont voici la signature :

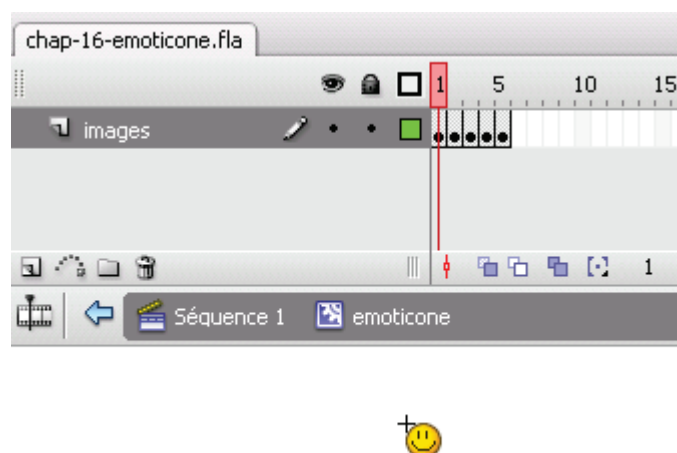
```
public function getCharBoundaries(charIndex:int):Rectangle
```

Celle-ci attend en paramètre l'index du caractère à rechercher et renvoie ses coordonnées sous la forme d'une instance de la classe `flash.geom.Rectangle`.

Grâce à cela, nous allons créer un système de gestion d'émoticônes.

Au sein d'un nouveau document Flash CS3, nous créons un nouveau symbole clip liée à une classe auto-générée `Emoticone`. Ce symbole contient différentes émoticônes sur chaque image le composant.

La figure 16-37 illustre les différentes images du clip :



*Figure 16-37. Symbole content les différentes émoticônes.*

Nous créons à présent un symbole clip contenant un champ texte nommé `contenuTexte` puis nous associons la classe `MoteurEmoticone` suivante par le panneau *Propriétés de liaison* :

```
package org.bytearray.emoticones
{
    import flash.display.Sprite;
    import flash.text.TextField;

    public class MoteurEmoticone extends Sprite
    {
        public var contenuTexte:TextField;

        public function MoteurEmoticone ()
        {

        }
    }
}
```



```

    }
}

```

Au sein du même document, nous instancions notre champ texte à gestion d’émoticônes à travers la classe de document suivante :

```

package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefault;
    import org.bytearray.emoticones.MoteurEmoticone;

    public class Document extends ApplicationDefault
    {

        public var champTexteEmoticone:MoteurEmoticone;

        public function Document ()

        {

            champTexteEmoticone = new MoteurEmoticone();

            champTexteEmoticone.x = (stage.stageWidth -
            champTexteEmoticone.width) / 2;
            champTexteEmoticone.y = (stage.stageHeight -
            champTexteEmoticone.height) / 2;

            addChild ( champTexteEmoticone );

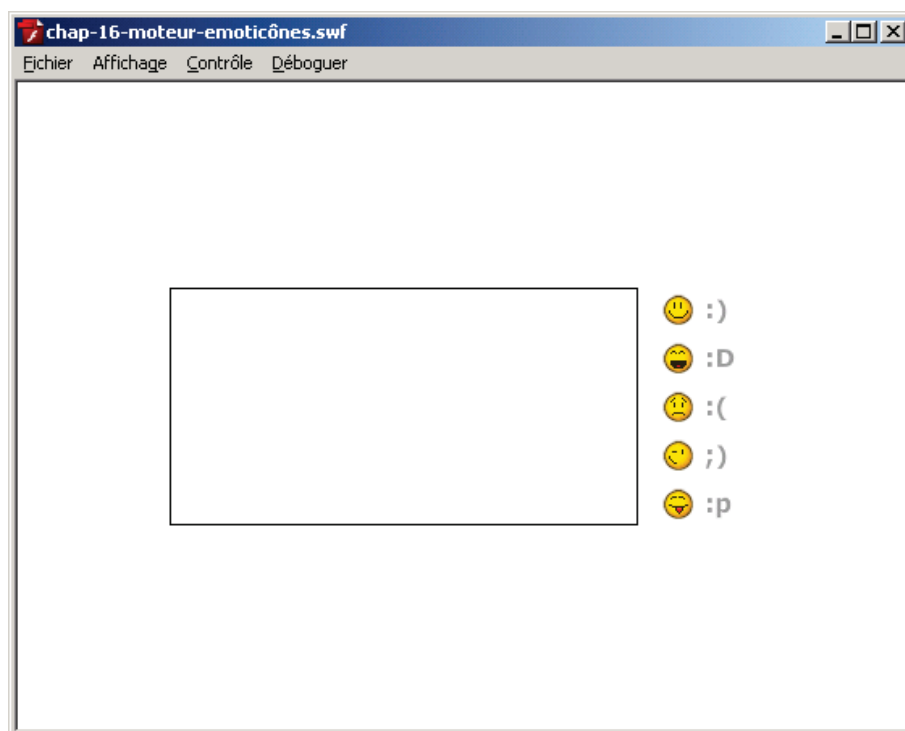
        }

    }

}

```

La figure 16-38 illustre le résultat :



*Figure 16-38. Affichage du moteur d'émoticônes.*

Afin d'écouter la saisie utilisateur et le défilement du contenu, nous écoutons les événements `Event.CHANGE` et `Event.SCROLL` auprès du champ texte :

```
package org.bytearray.emoticones
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.text.TextField;

    public class MoteurEmoticone extends Sprite
    {
        public var contenuTexte:TextField;

        public function MoteurEmoticone ()
        {
            contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
        );
            contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
        );
        }

        private function saisieUtilisateur ( pEvt:Event ):void
        {

```

```

        // affiche : texte saisie
        trace( pEvt.target.text );

    }

}

}

```

Lorsque du contenu est saisi dans le champ la méthode `saisieUtilisateur` est exécutée.

Afin de savoir si l'utilisateur a saisi une combinaison de touches correspondant à une émoticône, nous ajoutons un tableau nommé `codesTouches` contenant le code des combinaisons à saisir afin d'afficher une émoticône :

```

package org.bytearray.emoticones

{

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.text.TextField;

    public class MoteurEmoticone extends Sprite

    {

        public var contenuTexte:TextField;
        public var codesTouches:Array;
        public var lngTouches:int;

        public function MoteurEmoticone ()

        {

            codesTouches = new Array (":","D",":(",";"),":p");

            lngTouches = codesTouches.length

            contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
        );
            contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
        );

        }

        private function saisieUtilisateur ( pEvt:Event ):void

        {

            // affiche : texte saisie
            trace( pEvt.target.text );

        }

    }

}

```

```
| }
```

Au sein de la méthode `saisieUtilisateur` nous ajoutons une simple recherche de chaque combinaison à l'aide de la méthode `indexOf` de la classe `String` :

```
private function saisieUtilisateur ( pEvt:Event ):void
{
    var i:int;
    var j:int;

    for ( i = 0; i<lngTouches; i++ )
    {
        j = pEvt.target.text.indexOf ( codesTouches[i] );

        if ( j != -1 ) trace( "Combinaison trouvée à l'index : " + j );
    }
}
```

Si nous saisissons une combinaison contenue au sein du tableau `codeTouches` la variable `j` renvoie alors la position de la combinaison au sein du champ texte.

En cas de saisie multiple, seule la première combinaison est détectée car la méthode `indexOf` ne renvoie que la première chaîne trouvée.

Afin de corriger cela, nous ajoutons une boucle imbriquée afin de vérifier les multiples combinaisons existantes au sein du champ :

```
private function saisieUtilisateur ( pEvt:Event ):void
{
    var i:int;
    var j:int;

    for ( i = 0; i<lngTouches; i++ )
    {
        j = pEvt.target.text.indexOf ( codesTouches[i] );

        while ( j!= -1 )
        {
            trace( "Combinaison trouvée à l'index : " + j );

            j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );
        }
    }
}
```

Si nous testons le code précédent nous remarquons que les multiples combinaisons sont détectées.

Grâce à la méthode `getCharBoundaries` nous pouvons récupérer les coordonnées x et y de chaque combinaison au sein du champ `contenuTexte`.

Nous ajoutons une propriété `coordonnees` au sein de la classe, puis nous modifions la méthode `saisieUtilisateur` :

```
package org.bytearray.emoticones

{

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Rectangle;
    import flash.text.TextField;

    public class MoteurEmoticone extends Sprite

    {

        public var contenuTexte:TextField;
        public var codesTouches:Array;
        public var lngTouches:int;
        public var coordonnees:Rectangle;

        public function MoteurEmoticone ()

        {

            codesTouches = new Array (":","D",":(",";"),":p");

            lngTouches = codesTouches.length

            contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
        );
            contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
        );

        }

        private function saisieUtilisateur ( pEvt:Event ):void

        {

            var i:int;
            var j:int;

            for ( i = 0; i<lngTouches; i++ )
            {

                j = pEvt.target.text.indexOf ( codesTouches[i] );

                while ( j!= -1 )
                {

                    coordonnees = pEvt.target.getCharBoundaries ( j );
```

```
        y=2, w=7, h=18) // affiche : Combinaison trouvée à la position : (x=62,
        trace( "Combinaison trouvée à la position : " +
        coordonnees );

        j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

    }

}

}
```

La propriété `coordonnees` détermine la position de chaque combinaison de touches. Nous devons à présent interpréter ces coordonnées afin d’afficher l’émoticône correspondante.

Nous modifions à nouveau la méthode `saisieUtilisateur` en appelant la méthode `ajouteEmoticone` :

```
private function saisieUtilisateur ( pEvt:Event ):void
{
    var i:int;
    var j:int;

    for ( i = 0; i<lngTouches; i++ )
    {
        j = pEvt.target.text.indexOf ( codesTouches[i] );

        while ( j!= -1 )
        {
            coordonnees = pEvt.target.getCharBoundaries ( j );

            if ( coordonnees != null ) ajouteEmoticone ( coordonnees, i
        );

            j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

        }

    }
}
```

Puis nous définissons une propriété `icone` de type `Emoticone` ainsi que la méthode `ajouteEmoticone` :

```
private function ajouteEmoticone ( pRectangle:Rectangle, pIndex:int
):Emoticone
{
```

```
icone = new Emoticone();
icone.gotoAndStop ( pIndex + 1 );

icone.x = pRectangle.x + 1;
icone.y = pRectangle.y - ((contenuTexte.scrollV -
1)*(contenuTexte.textHeight/contenuTexte.numLines))+1;

addChild ( icone );

return icone;

}
```

Si nous testons le code actuel, nous remarquons que les émoticônes sont affichées correctement. Si nous revenons en arrière lors de la saisie nous devons supprimer les émoticônes déjà présentes afin de ne pas les conserver à l’affichage.

Nous devons définir une propriété `tableauEmoticones` contenant la référence de chaque émoticône ajouté, puis créer un nouveau tableau au sein du constructeur :

```
public function MoteurEmoticone ()
{

    tableauEmoticones = new Array();

    codesTouches = new Array ( ":" , ":"D" , ":( , "; ) , ":"p" );
    lngTouches = codesTouches.length;

    contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur );
    contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur );

}
```

A chaque saisie utilisateur nous supprimons toutes les émoticônes créées auparavant afin de nettoyer l’affichage :

```
private function saisieUtilisateur ( pEvt:Event ):void
{

    var i:int;
    var j:int;

    var nombreEmoticones:int = tableauEmoticones.length;

    for ( i = 0; i< nombreEmoticones; i++ ) removeChild
(tableauEmoticones[i] );

    tableauEmoticones = new Array();

    for ( i = 0; i<lngTouches; i++ )
    {

        j = pEvt.target.text.indexOf ( codesTouches[i] );
```

```
        while ( j!= -1 )
        {

            coordonnees = pEvt.target.getCharBoundaries ( j );

            if ( coordonnees != null ) tableauEmoticones.push (
ajouteEmoticone ( coordonnees, i ) );

            j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

        }

    }

}
```

Voici le code complet de la classe **MoteurEmoticone** :

```
package org.bytearray.emoticones

{

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.geom.Rectangle;
    import flash.text.TextField;

    public class MoteurEmoticone extends Sprite

    {

        public var contenuTexte:TextField;
        public var codesTouches:Array;
        public var lngTouches:int;
        public var coordonnees:Rectangle;
        public var icone:Emoticone;
        public var tableauEmoticones:Array;

        public function MoteurEmoticone ()

        {

            tableauEmoticones = new Array();

            codesTouches = new Array ( ":" , ":"D" , ":( , "; ) , ":"p" );

            lngTouches = codesTouches.length;

            contenuTexte.addEventListener ( Event.CHANGE, saisieUtilisateur
);
            contenuTexte.addEventListener ( Event.SCROLL, saisieUtilisateur
);

        }

        private function saisieUtilisateur ( pEvt:Event ):void

        {

            var i:int;
            var j:int;
```



```
        var nombreEmoticones:int = tableauEmoticones.length;

        for ( i = 0; i< nombreEmoticones; i++ ) removeChild
(tableauEmoticones[i] );

        tableauEmoticones = new Array();

        for ( i = 0; i<lngTouches; i++ )
        {

            j = pEvt.target.text.indexOf ( codesTouches[i] );

            while ( j!= -1 )
            {

                coordonnees = pEvt.target.getCharBoundaries ( j );

                if ( coordonnees != null ) tableauEmoticones.push (
ajouteEmoticone ( coordonnees, i ) );

                j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );

            }

        }

    }

    private function ajouteEmoticone ( pRectangle:Rectangle, pIndex:int
):Emoticone

    {

        icone = new Emoticone();
        icone.gotoAndStop ( pIndex + 1 );

        icone.x = pRectangle.x + 1;
        icone.y = pRectangle.y - ((contenuTexte.scrollV -
1)*(contenuTexte.textHeight/contenuTexte.numLines))+1;

        addChild ( icone );

        return icone;

    }

}
```

La classe `MoteurEmoticone` peut ainsi être intégrée à un chat connecté par `XMLSocket` ou Flash Media Server, nous la réutiliserons au cours du chapitre 18 intitulé *Sockets*.

D'autres émoticônes pourraient être intégrées, ainsi que d'autres fonctionnalités.

Afin d'aller plus loin, nous allons créer un éditeur de texte enrichi. Celui-ci nous permettra de mettre en forme du texte de manière simplifiée.

## A retenir

- La méthode `getCharBoundaries` permet de connaître la position exacte d'un caractère au sein d'un champ texte en retournant un objet `Rectangle`.

## Créer un éditeur de texte

Nous avons très souvent besoin d'éditer du texte de manière avancée au sein d'applications Flash. La classe `TextField` a été grandement enrichie en ActionScript 3. De nombreuses méthodes et propriétés facilitent le développement d'applications ayant recours au texte.

Afin de mettre en application les notions précédemment étudiées, nous allons développer à présent un éditeur de texte.

La figure 16-39 illustre l'éditeur en question :



*Figure 16-39. Editeur de texte.*

Nous allons utiliser pour sa conception les différentes propriétés et méthodes listées ci-dessous :

- `setTextFormat` : la méthode `setTextFormat` nous permet d'affecter un style particulier à une sélection ;
- `getTextFormat` : la méthode `getTextFormat` nous permet de récupérer un style existant lié à une sélection afin de le modifier ;
- `selectionBeginIndex` : la propriété `selectionBeginIndex` permet de connaître le début de sélection du texte ;
- `selectionEndIndex` : la propriété `selectionEndIndex` permet de connaître la fin de sélection du texte ;
- `alwaysShowSelection` : la propriété `alwaysShowSelection` permet de conserver le texte sélectionné bien que la souris clique sur un autre élément interactif ;

Nous commençons par créer la classe `EditeurTexte` suivante au sein du paquetage `org.bytearray.richtext` :

```
package org.bytearray.texte.editeur
{
    import flash.display.Sprite;
    import flash.events.Event;
```

```

public class EditeurTexte extends Sprite
{
    public function EditeurTexte ()
    {
        addEventListener ( Event.ADDED_TO_STAGE, activation );
        addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
    }

    private function activation ( pEvt:Event ):void
    {
        trace("activation");
    }

    private function desactivation ( pEvt:Event ):void
    {
        trace("desactivation");
    }
}

```

Nous écoutons les événements `Event.ADDED_TO_STAGE` et `Event.REMOVED_FROM_STAGE` afin de gérer l'activation et la désactivation de l'éditeur.

Sur la scène nous créons un clip contenant les différents boutons graphiques comme l'illustre la figure 16-40 :



*Figure 16-40. Editeur de texte enrichi.*

Grace au panneau *Propriétés de liaison*, nous lions ce clip à notre classe `EditeurTexte` précédemment définie.

Nous ajoutons les écouteurs auprès des différents boutons de l'interface au sein de la classe `EditeurTexte` :

```

package org.bytearray.texte.editeur
{

```

```
import flash.display.Sprite;
import flash.display.SimpleButton;
import flash.text.TextField;
import flash.events.Event;
import flash.events.MouseEvent;
import fl.controls.ComboBox;
import fl.controls.ColorPicker;
import fl.events.ColorPickerEvent;

public class EditeurTexte extends Sprite
{
    public var boutonAlignerGauche:SimpleButton;
    public var boutonAlignerCentre:SimpleButton;
    public var boutonAlignerDroite:SimpleButton;
    public var boutonAlignerJustifier:SimpleButton;
    public var boutonCrenage:SimpleButton;
    public var boutonGras:SimpleButton;
    public var boutonItalique:SimpleButton;
    public var boutonSousLigne:SimpleButton;
    public var boutonLien:SimpleButton;
    public var champLien:TextField;
    public var listeDeroulantesPolices:ComboBox;
    public var nuancier:ColorPicker;

    public function EditeurTexte ()
    {
        addEventListener ( Event.ADDED_TO_STAGE, activation );
        addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
    }

    private function activation ( pEvt:Event ):void
    {
        boutonGras.addEventListener( MouseEvent.CLICK, affecteGras );
        boutonItalique.addEventListener( MouseEvent.CLICK,
affecteItalique );
        boutonSousLigne.addEventListener( MouseEvent.CLICK,
affecteSousLigne );

        boutonAlignerGauche.addEventListener( MouseEvent.CLICK,
alignerGauche );
        boutonAlignerCentre.addEventListener( MouseEvent.CLICK,
alignerCentre );
        boutonAlignerDroite.addEventListener( MouseEvent.CLICK,
alignerDroite );
        boutonAlignerJustifier.addEventListener( MouseEvent.CLICK,
alignerJustifier );
        nuancier.addEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
        boutonCrenage.addEventListener ( MouseEvent.CLICK, affecteCrenage
);
        listeDeroulantesPolices.addEventListener ( Event.CHANGE,
affectePolice );
        boutonLien.addEventListener ( MouseEvent.CLICK, affecteLien );
    }
}
```

```
private function desactivation ( pEvt:Event ):void
{
    boutonGras.removeEventListener( MouseEvent.CLICK, affecteGras );
    boutonItalique.removeEventListener( MouseEvent.CLICK,
affecteItalique );
    boutonSousLigne.removeEventListener( MouseEvent.CLICK,
affecteSousLigne );

    boutonAlignerGauche.removeEventListener( MouseEvent.CLICK,
alignerGauche );
    boutonAlignerCentre.removeEventListener( MouseEvent.CLICK,
alignerCentre );
    boutonAlignerDroite.removeEventListener( MouseEvent.CLICK,
alignerDroite );
    boutonAlignerJustifier.removeEventListener( MouseEvent.CLICK,
alignerJustifier );
    nuancier.removeEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
    boutonCrenage.removeEventListener ( MouseEvent.CLICK,
affecteCrenage );
    listeDeroulantesPolices.removeEventListener ( Event.CHANGE,
affectePolice );
    boutonLien.removeEventListener ( MouseEvent.CLICK, affecteLien );
}

function affectePolice ( pEvt:Event ):void
{
}

function affecteCrenage ( pEvt:MouseEvent ):void
{
}

function alignerJustifier ( pEvt:MouseEvent ):void
{
}

function couleurSelection ( pEvt:ColorPickerEvent ):void
{
}

function alignerGauche ( pEvt:MouseEvent ):void
{
}

function alignerCentre ( pEvt:MouseEvent ):void
{
}
```

```
    }

    function alignerDroite ( pEvt:MouseEvent ):void
    {
    }

    function affecteGras ( pEvt:MouseEvent ):void
    {
    }

    function affecteItalique ( pEvt:MouseEvent ):void
    {
    }

    function affecteSousLigne ( pEvt:MouseEvent ):void
    {
    }

    private function affecteLien ( pEvt:MouseEvent ):void
    {
    }
}
}
```

Nous remplissons la liste déroulante en énumérant les polices installées. Nous définissons une méthode `affichePolices` :

```
private function affichePolices ():void
{
    var polices:Array = Font.enumerateFonts(true);
    var donnees:Array = new Array();

    for ( var p:String in polices ) donnees.push ( { label :
polices[p].fontName, data : polices[p].fontName } );

    listeDeroulantesPolices.dataProvider = new DataProvider ( donnees );
}
```

Puis nous importons les classes nécessaires :

```
import flash.text.Font;
import fl.data.DataProvider;
```

Nous modifions la méthode `activation` en appelant la méthode `affichePolices` :

```

private function activation ( pEvt:Event ):void
{
    boutonGras.addEventListener( MouseEvent.CLICK, affecteGras );
    boutonItalique.addEventListener( MouseEvent.CLICK, affecteItalique );
    boutonSousLigne.addEventListener( MouseEvent.CLICK, affecteSousLigne
);

    boutonAlignerGauche.addEventListener( MouseEvent.CLICK, alignerGauche
);
    boutonAlignerCentre.addEventListener( MouseEvent.CLICK, alignerCentre
);
    boutonAlignerDroite.addEventListener( MouseEvent.CLICK, alignerDroite
);
    boutonAlignerJustifier.addEventListener( MouseEvent.CLICK,
alignerJustifier );
    nuancier.addEventListener ( ColorPickerEvent.CHANGE, couleurSelection
);
    boutonCrenage.addEventListener ( MouseEvent.CLICK, affecteCrenage );
    listeDeroulantesPolices.addEventListener ( Event.CHANGE, affectePolice
);
    boutonLien.addEventListener ( MouseEvent.CLICK, affecteLien );

    affichePolices();
}

```

Si nous testons notre application, la liste déroulante de l'éditeur enrichi doit afficher la totalité des polices présentes sur la machine client comme l'illustre la figure 16-41 :



*Figure 16-41. Editeur de texte enrichi.*

Afin de stocker le style en cours nous définissons une propriété `formatEnCours` :

```
private var formatEnCours:TextFormat;
```

Pour cela, nous ajoutons une propriété `champCible` :

```
private var champCible:TextField;
```

Puis une méthode `cible` permettant de spécifier le champ texte à enrichir :

```

public function cible ( pChamp:TextField ):void
{
    if ( pChamp != champCible )
    {
        champCible = pChamp;
        champCible.alwaysShowSelection = true;
    }
}

```

```
}  
}
```

La logique de l'éditeur est quasi complète, il ne nous reste plus qu'à définir la logique nécessaire au sein de chaque fonction d'affectation de style.

Voici le code complet de la classe `EditeurTexte` :

```
package org.bytearray.texte.editeur  
  
{  
  
    import flash.display.Sprite;  
    import flash.display.SimpleButton;  
    import flash.text.TextField;  
    import flash.text.TextFormat;  
    import flash.text.TextFormatAlign;  
    import flash.events.Event;  
    import flash.events.MouseEvent;  
    import fl.controls.ComboBox;  
    import fl.controls.ColorPicker;  
    import fl.events.ColorPickerEvent;  
    import flash.text.Font;  
    import fl.data.DataProvider;  
  
    public class EditeurTexte extends Sprite  
    {  
  
        public var boutonAlignerGauche:SimpleButton;  
        public var boutonAlignerCentre:SimpleButton;  
        public var boutonAlignerDroite:SimpleButton;  
        public var boutonAlignerJustifier:SimpleButton;  
        public var boutonCrenage:SimpleButton;  
        public var boutonGras:SimpleButton;  
        public var boutonItalique:SimpleButton;  
        public var boutonSousLigne:SimpleButton;  
        public var boutonLien:SimpleButton;  
        public var champLien:TextField;  
        public var listeDeroulantesPolices:ComboBox;  
        public var nuancier:ColorPicker;  
  
        private var formatEnCours:TextFormat;  
        private var champCible:TextField;  
  
        public function EditeurTexte ()  
        {  
  
            addEventListener ( Event.ADDED_TO_STAGE, activation );  
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );  
  
        }  
  
        private function activation ( pEvt:Event ):void  
        {  
  
            boutonGras.addEventListener( MouseEvent.CLICK, affecteGras );  
  
        }  
    }  
}
```



```

        boutonItalique.addEventListener( MouseEvent.CLICK,
affecteItalique );
        boutonSousLigne.addEventListener( MouseEvent.CLICK,
affecteSousLigne );

        boutonAlignerGauche.addEventListener( MouseEvent.CLICK,
alignerGauche );
        boutonAlignerCentre.addEventListener( MouseEvent.CLICK,
alignerCentre );
        boutonAlignerDroite.addEventListener( MouseEvent.CLICK,
alignerDroite );
        boutonAlignerJustifier.addEventListener( MouseEvent.CLICK,
alignerJustifier );
        nuancier.addEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
        boutonCrenage.addEventListener ( MouseEvent.CLICK, affecteCrenage
);
        listeDeroulantesPolices.addEventListener ( Event.CHANGE,
affectePolice );
        boutonLien.addEventListener ( MouseEvent.CLICK, affecteLien );

        affichePolices();

    }

    public function cible ( pChamp:TextField ):void

    {

        if ( pChamp != champCible )
        {
            champCible = pChamp;
            champCible.alwaysShowSelection = true;
        }

    }

    private function desactivation ( pEvt:Event ):void

    {

        boutonGras.removeEventListener( MouseEvent.CLICK, affecteGras );
        boutonItalique.removeEventListener( MouseEvent.CLICK,
affecteItalique );
        boutonSousLigne.removeEventListener( MouseEvent.CLICK,
affecteSousLigne );

        boutonAlignerGauche.removeEventListener( MouseEvent.CLICK,
alignerGauche );
        boutonAlignerCentre.removeEventListener( MouseEvent.CLICK,
alignerCentre );
        boutonAlignerDroite.removeEventListener( MouseEvent.CLICK,
alignerDroite );
        boutonAlignerJustifier.removeEventListener( MouseEvent.CLICK,
alignerJustifier );
        nuancier.removeEventListener ( ColorPickerEvent.CHANGE,
couleurSelection );
        boutonCrenage.removeEventListener ( MouseEvent.CLICK,
affecteCrenage );
        listeDeroulantesPolices.removeEventListener ( Event.CHANGE,
affectePolice );
        boutonLien.removeEventListener ( MouseEvent.CLICK, affecteLien );
    }

```

```
    }

    private function affichePolices ():void
    {
        var polices:Array = Font.enumerateFonts(true);
        var donnees:Array = new Array();

        for (var p in polices ) donnees.push ( { label :
polices[p].fontName, data : polices[p].fontName } );

        listeDeroulantesPolices.dataProvider = new DataProvider
(donnees);
    }

    function affecteGras ( pEvt:MouseEvent ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.bold = !formatEnCours.bold;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function affecteItalique ( pEvt:MouseEvent ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.italic = !formatEnCours.italic;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function affecteSousLigne ( pEvt:MouseEvent ):void
    {
```

```
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.underline = !formatEnCours.underline;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function alignerGauche ( pEvt:MouseEvent ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.LEFT ) ? TextFormatAlign.LEFT : TextFormatAlign.LEFT;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function alignerCentre ( pEvt:MouseEvent ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.CENTER ) ? TextFormatAlign.CENTER : TextFormatAlign.LEFT;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function alignerDroite ( pEvt:MouseEvent ):void
    {
```

```
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.RIGHT ) ? TextFormatAlign.RIGHT : TextFormatAlign.LEFT;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function alignerJustifier ( pEvt:MouseEvent ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.align = ( formatEnCours.align !=
TextFormatAlign.JUSTIFY ) ? TextFormatAlign.JUSTIFY : TextFormatAlign.LEFT;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function couleurSelection ( pEvt:ColorPickerEvent ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.color = pEvt.color;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function affecteCrenage ( pEvt:MouseEvent ):void
    {
```

```
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.kerning = !formatEnCours.kerning;

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    function affectePolice ( pEvt:Event ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.font = ( formatEnCours.font !=
pEvt.target.selectedItem.data ) ? pEvt.target.selectedItem.data : "Verdana";

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }

    private function affecteLien ( pEvt:MouseEvent ):void
    {
        if ( champCible != null )
        {
            formatEnCours = champCible.getTextFormat(
champCible.selectionBeginIndex, champCible.selectionEndIndex );

            formatEnCours.url = formatEnCours.url == "" ? champLien.text
: "";

            champCible.setTextFormat( formatEnCours,
champCible.selectionBeginIndex, champCible.selectionEndIndex );

        } else throw new Error ("Le champ cible n'a pas été défini.");
    }
}

}
```

Afin de tester notre éditeur, nous associons la classe de document suivante :

```
package org.bytearray.document
{
    import flash.display.Sprite;
    import flash.text.TextField;
    import org.bytearray.abstrait.ApplicationDefault;
    import org.bytearray.texte.editeur.EditeurTexte;

    public class Document extends ApplicationDefault
    {
        private var editeur:EditeurTexte;
        public var texteCible:TextField;

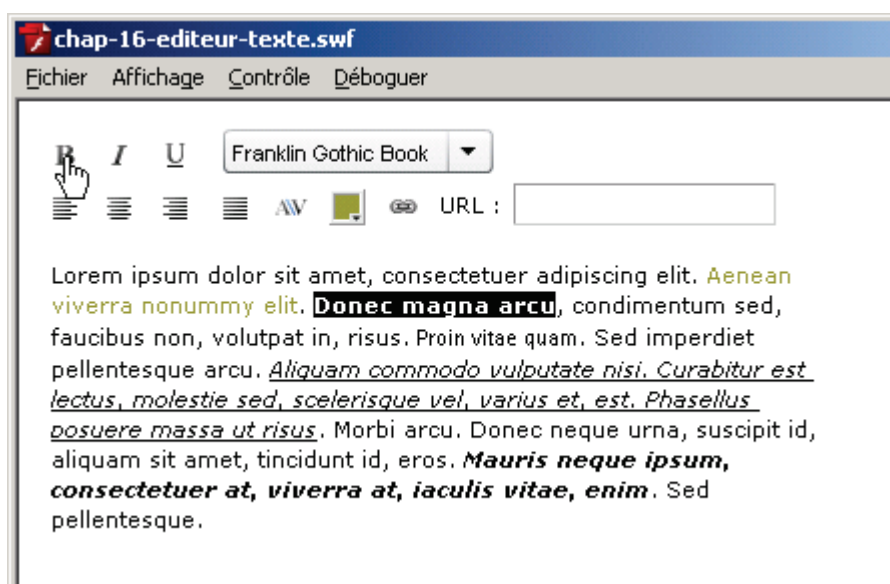
        public function Document ()
        {
            // instantiation de l'éditeur enrichi
            editeur = new EditeurTexte();

            // positionnement
            editeur.x = editeur.y = 15;

            // ajout à l'affichage
            addChild ( editeur );

            // affectation du champ texte cible
            editeur.cible ( texteCible );
        }
    }
}
```

La figure 16-42 illustre l'éditeur de texte en action :



*Figure 16-42. Editeur de texte finalisé.*

Il convient de s'attarder quelques instants sur la propriété `alwaysShowSelection` de la classe `TextField`.

ActionScript 3 introduit cette propriété facilitant grandement le développement d'un éditeur comme celui que nous venons de terminer.

La puissance de cette propriété réside dans la conservation de la sélection du texte bien que l'utilisateur entre en interaction avec d'autres éléments graphiques. Cela nous permet de récupérer facilement la sélection en cours à l'aide des propriétés `selectionBeginIndex` et `selectionEndIndex` et d'affecter le style voulu.

D'autres fonctionnalités pourraient être ajoutées à l'éditeur, nous pourrions imaginer un export de la mise en forme du contenu texte sous la forme de feuille de style CSS, ou encore d'autres fonctionnalités liées à la mise en forme.

## A retenir

- Les propriétés `selectionBeginIndex` et `selectionEndIndex` nous permettent de connaître la selection en cours.
- Grace aux méthodes `getTextFormat` et `setTextFormat`, nous pouvons récupérer le style d'une partie du texte, le modifier et l'affecter à nouveau.
- La propriété `alwaysShowSelection` permet de conserver la selection du texte, même si la souris entre en interaction avec d'autres objets interactifs.

Au cours du prochain chapitre nous découvrirons les nouvelles fonctionnalités apportées par ActionScript 3 en matière de son et de vidéo.



# 17

## Son et vidéo

<b>LECTURE DE SONS .....</b>	<b>1</b>
LIRE UN SON PROVENANT DE LA BIBLIOTHEQUE .....	2
LIRE UN SON DYNAMIQUE .....	4
LA CLASSE SOUNDLOADERCONTEXT .....	7
TRANSFORMATION DU SON .....	9
MODIFICATION GLOBALE DU SON .....	30
LIRE LE SPECTRE D'UN SON .....	31
TRANSFORMÉE DE FOURIER .....	55
LE FORMAT MPEG-4 AUDIO .....	58
<b>LA VIDEO DANS FLASH .....</b>	<b>62</b>
LE FORMAT MPEG-4 VIDEO .....	63
LA CLASSE VIDEO .....	65
TRANSFORMATION DU SON LIE A UN OBJET NETSTREAM .....	69
MODE PLEIN-ECRAN .....	70

### Lecture de sons

Le son fut d'abord considéré comme un élément secondaire sur le web. La situation s'est inversée peu à peu jusqu'à aujourd'hui où le lecteur Flash figure parmi les principaux promoteurs du son. De nombreuses sociétés ont d'ailleurs choisi le lecteur Flash comme plateforme de diffusion ou de lecture.

Nous allons découvrir au cours de ce chapitre comment utiliser les différentes classes liées au son et à la vidéo en ActionScript 3, afin d'intégrer au mieux ces supports dans nos applications Flash.

Nous allons nous attarder dès à présent sur les différentes classes liées au son dans Flash :

- `flash.media.Sound` : la classe `Sound` permet la lecture de sons.

- `flash.media.SoundTransform` : la classe `SoundTransform` permet de modifier le son (volume, balance).
- `flash.media.SoundChannel` : la classe `SoundChannel` permet de contrôler le son. Chaque son en cours de lecture est associé à un objet `SoundChannel`.
- `flash.media.LoaderContext` : la classe `LoaderContext` est liée au préchargement et au modèle de sécurité du lecteur Flash.
- `flash.media.SoundMixer` : la classe `SoundMixer` offre un contrôle global sur les sons en cours de lecture.
- `flash.net.NetConnection` : la classe `NetConnection` est utilisée pour le chargement de fichiers audio MPEG-4.
- `flash.net.NetStream` : la classe `NetStream` est utilisée pour la manipulation de fichiers audio MPEG-4.

La liste peut paraître longue, mais nous verrons que leur utilisation s'avère d'une étonnante simplicité.

## Lire un son provenant de la bibliothèque

Afin d'entamer notre aventure, nous allons commencer par lire un son provenant de la bibliothèque. Pour cela, nous utilisons le raccourci clavier CTRL+R ou l'option *Importer dans la bibliothèque*.

Par le panneau *Liaisons*, nous associons le son en bibliothèque à une classe auto générée `Rythmique`, celle-ci étend la classe `Sound` :

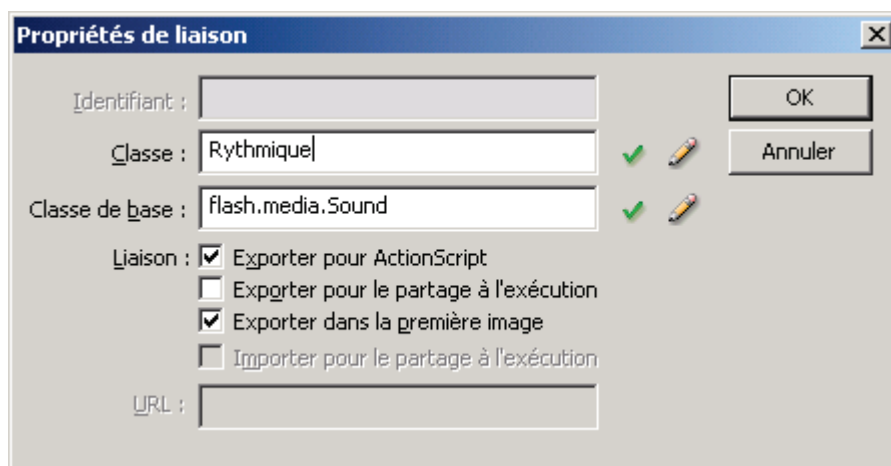


Figure 17-1. Son associée à la classe `Rythmique` auto générée.

Le son est instancié à l'aide de l'opérateur `new` :

```
// instanciation du son
var rythme:Rythmique = new Rythmique();
```

Afin de démarrer la lecture du son créé, nous utilisons la méthode `play` dont voici la signature :

```
public function play(startTime:Number = 0, loops:int = 0,
sndTransform:SoundTransform = null):SoundChannel
```

Celle-ci accepte trois paramètres :

- `startTime` : la position en milli secondes à partir de laquelle la lecture doit commencer.
- `loops` : nombre de boucles.
- `sndTransform` : un objet de transformation associé au son joué. Nous reviendrons très vite sur la classe `SoundTransform`.

Contrairement à la classe `Sound` présente en ActionScript 1 et 2, la classe `Sound` ne dispose pas en ActionScript 3 de méthode `stop`.

L'appel de la méthode `play` affecte le son joué à un canal audio, et retourne un objet de type `SoundChannel` représentant ce canal. C'est grâce à ce dernier que nous pouvons contrôler le son et l'arrêter.

---

Notons que le lecteur Flash 9 peut lire désormais jusqu'à 32 canaux simultanés.

---

Dans le code suivant, nous lisons le son depuis le départ une seule fois seulement :

```
// instantiation du son
var rythme:Rythmique = new Rythmique();

// lecture du son et récupération de l'objet SoundChannel associé au son
var canalRythme:SoundChannel = rythme.play();
```

En spécifiant le paramètre `startTime`, nous pouvons décaler le point de départ de la lecture du son :

```
var canalRythme:SoundChannel = rythme.play( 6000 );
```

Le son est ainsi lu à partir de la sixième seconde. Par défaut, le son est joué une seule fois, mais nous pouvons spécifier un nombre de boucles grâce au deuxième paramètre `loops` :

```
var canalRythme:SoundChannel = rythme.play( 6000, 2 );
```

Notons que si un nombre de boucles est spécifié ainsi qu'une position de départ, celle-ci est conservée lors de la boucle.

---

Lors de la lecture en boucle d'un son, la propriété `position` de l'objet `SoundChannel` ne se réinitialise pas à zéro. Pour un son d'une durée de 5000 milli-secondes lu en boucle 3 fois, celle-ci vaut 15 000 ms en

---

fin de lecture. Il s'agit d'un bogue du lecteur Flash 9.0.115.

L'objet `SoundChannel` étant renvoyé par l'appel de la méthode `play`, il n'est pas possible d'étendre la classe `SoundChannel` afin de corriger ce bogue.

---

A l'aide de la méthode `stop` définie par la classe `SoundChannel`, nous stoppons le son lorsque l'utilisateur clique sur la scène :

```
// instantiation du son
var rythme:Rythmique = new Rythmique();

// lecture du son et récupération de l'objet SoundChannel associé au son
var canalRythme:SoundChannel = rythme.play( 6000, 2 );

stage.addEventListener( MouseEvent.CLICK, stoppeSon );

function stoppeSon ( pEvt:MouseEvent ):void
{
    // le son est coupé
    canalRythme.stop();
}
```

Afin de garantir un poids minimum et d'assurer un chargement à la demande, il peut être intéressant de charger dynamiquement les sons de l'application, c'est ce que nous allons voir dans cette nouvelle partie.

## A retenir

- La méthode `play` démarre la lecture du son et l'associe à un canal audio représenté par un objet `SoundChannel`.
- Le lecteur Flash 9 permet la lecture de 32 sons simultanés.
- La lecture du son associé au canal est interrompue grâce à la méthode `stop` de l'objet `SoundChannel`.

## Lire un son dynamique

Le chargement de son peut être réalisé de manière dynamique. Les fichiers audio résident à l'extérieur de l'animation et sont chargés dynamiquement.

Deux techniques peuvent être employées :

La première consiste à créer un objet `URLRequest` valide pointant vers le fichier MP3 et passer ce dernier au constructeur de la classe `Sound` :

```
// création d'un objet Sound, la méthode load est automatiquement appelée  
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );
```

Si la classe `Sound` détecte un objet `URLRequest` valide, le son est chargé automatiquement au sein du lecteur à l'aide de la méthode `load`. Le cas échéant, l'appel de la méthode `load` est obligatoire pour démarrer le chargement du son.

La méthode `load` possède la signature suivante :

```
public function load(stream:URLRequest, context:SoundLoaderContext = null):void
```

Voici le détail de chacun des paramètres :

- `stream` : un objet `URLRequest` pointant vers le fichier MP3 à charger.
- `context` : un objet `SoundLoaderContext` spécifiant la durée de préchargement en mémoire tampon ainsi que des consignes liées au chargement de fichiers de régulation.

Attention, contrairement à l'intégration de son au sein de la bibliothèque compatible avec la plupart des formats audio, la méthode `load` de la classe `Sound` permet uniquement le chargement de fichiers MP3. Si nous tentons de charger un autre format de fichiers audio, aucune erreur spécifique n'est levée, le son n'est pas joué.

La seconde technique consiste à créer l'objet `Sound` puis appeler manuellement la méthode `load`.

Dans le code suivant nous ne passons pas d'objet `URLRequest` au constructeur de la classe `Sound`, le son est chargé à l'aide de la méthode `load` :

```
// création d'un objet Sound  
var monSon:Sound = new Sound();  
  
// chargement dynamique du son  
monSon.load ( new URLRequest ("son.mp3") );
```

Afin de gérer le chargement de son dynamique, l'objet `Sound` diffuse différents événements dont voici le détail :

- `Event.COMPLETE` : diffusé lorsque le chargement du son est terminé.
- `Event.ID3` : diffusé lorsque les informations ID3 sont disponibles.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque le chargement du son échoue.
- `Event.OPEN` : diffusé lorsque le lecteur commence à charger le son.
- `ProgressEvent.PROGRESS` : diffusé lorsque le chargement est en cours. Celui-ci renseigne sur le nombre d'octets chargés et totaux.

Dans le code suivant nous écoutons chacun des événements :

```
// instantiation d'un objet Sound
var son:Sound = new Sound();

// chargement dynamique du son
son.load ( new URLRequest ("son.mp3") );

// écoute des différents événements
son.addEventListener( Event.OPEN, chargementDemarre );
son.addEventListener( Event.ID3, informationsID3 );
son.addEventListener( ProgressEvent.PROGRESS, chargementEnCours );
son.addEventListener( Event.COMPLETE, chargementTermine );
son.addEventListener( IOErrorEvent.IO_ERROR, erreurChargement );

function chargementDemarre ( pEvt:Event ):void
{
    trace("chargement démarré");
}

function informationsID3 ( pEvt:Event ):void
{
    trace("informations ID3");
}

function chargementEnCours ( pEvt:ProgressEvent ):void
{
    trace("chargement en cours : " + pEvt.bytesLoaded + " / " + pEvt.bytesTotal
);
}

function chargementTermine ( pEvt:Event ):void
{
    trace("chargement terminé");
}

function erreurChargement ( pEvt:IOErrorEvent ):void
{
    trace("erreur de chargement");
}
```

Quelle que soit la technique employée pour charger dynamiquement un son, sa lecture doit être initiée à l'aide de la méthode `play`.

Si celle-ci est appelée en même temps que la méthode `load`, la lecture du son est entamée lorsque suffisamment de données audio ont été téléchargées.

Si nous tentons de démarrer la lecture du son avant l'avoir chargé, une erreur de type `ArgumentError` est levée :

```
| ArgumentError: Error #2068: Son non valide
```

Attention, dans un contexte de chargement dynamique, la méthode `load` ne renvoie pas d'objet `SoundChannel`.

Seule la méthode `play` le permet :

```
// instantiation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son, la méthode play retourne un objet SoundChannel
var canalSon:SoundChannel = son.play();
```

Il est important de noter que le chargement dynamique de sons est régit par le modèle de sécurité du lecteur.

Par défaut, le chargement et la lecture d'un fichier son provenant d'un domaine différent est autorisée, mais l'accès aux données du fichier son est régulée.

Dans un contexte inter-domaine, l'utilisation de la propriété `id3` de l'objet `Sound`, de la méthode `computeSpectrum`, de la classe `SoundMixer` ou de l'objet `SoundTransform` lève une erreur de type `SecurityError`.

Il convient alors d'utiliser un fichier de régulation sur le domaine distant afin d'autoriser la manipulation du son.

Rappelez-vous que le lecteur Flash ne charge pas automatiquement de fichier de régulation afin de limiter la bande passante utilisée.

Nous avons vu au cours du chapitre 13 intitulé *Chargement de contenu* que la classe `LoaderContext` permettait de spécifier si le lecteur Flash devait tenter de charger un fichier de régulation.

Une classe équivalente nommée `SoundLoaderContext` existe au sein du paquetage `flash.media` dans le cas de chargement de son dynamique.

## A retenir

- Le chargement d'un son dynamique est assuré par la méthode `load` de la classe `Sound`.
- Si la méthode `play` est appelée après l'appel de la méthode `load`, la lecture du son démarre lorsque le lecteur Flash a chargé suffisamment de données.

## La classe `SoundLoaderContext`

Lorsqu'un son est chargé depuis un domaine différent, celui-ci peut seulement être chargé et lu. Afin d'extraire des informations du son, ou d'utiliser des méthodes telles `computeSpectrum` de la classe

`SoundMixer`, un fichier de régulation doit être placé sur le serveur l'hébergeant afin d'autoriser le SWF ayant initié le chargement.

Nous avons utilisé au cours du chapitre 13 la classe `LoaderContext` afin d'indiquer au lecteur Flash de charger un fichier de régulation.

Dans le cas de chargement de fichiers audio, nous devons utiliser la classe `SoundLoaderContext` dont voici la signature du constructeur :

```
public function SoundLoaderContext(bufferTime:Number = 1000,  
checkPolicyFile:Boolean = false)
```

Voici le détail de chaque paramètre :

- `bufferTime` : le paramètre `bufferTime` permet de définir le temps de préchargement en mémoire tampon avant que la lecture du son ne démarre. La valeur par défaut est de 1000 milli-secondes. Cette fonctionnalité permet de charger à l'avance quelques secondes du flux afin d'assurer une lecture ininterrompue en cas de problème de connexion.
- `checkPolicyFile` : en passant la valeur `true` au paramètre `checkPolicyFile` nous demandons au lecteur Flash de télécharger un fichier de régulation à la racine du serveur hébergeant le fichier audio.

Dans le code suivant, nous chargeons un son hébergé sur un domaine distant, nous forçons le téléchargement d'un fichier de régulation :

```
// création d'un objet son et chargement d'un mp3  
var monSon:Sound = new Sound ();  
  
// création d'un objet SoundLoaderContext  
// 5 secondes sont mises en mémoire tampon  
// le lecteur Flash tente de charger un fichier de régulation  
var contexteAudio:SoundLoaderContext = new SoundLoaderContext ( 5000, true );  
  
// chargement d'un son auprès d'un domaine distant  
monSon.load ( new URLRequest ("http://www.monDomaineDistant.org/son.mp3") );  
  
// lecture du son  
var canalSon:SoundChannel = monSon.play();
```

Au lieu d'initialiser l'objet `SoundLoaderContext` par les paramètres du constructeur, les propriétés équivalentes peuvent être utilisées :

```
// création d'un objet SoundLoaderContext  
var contexteAudio:SoundLoaderContext = new SoundLoaderContext ();  
  
// 5 secondes sont mises en mémoire tampon  
contexteAudio.bufferTime = 5000;  
  
// le lecteur Flash tente de charger un fichier de régulation  
contexteAudio.checkPolicyFile = true;  
  
// chargement d'un son auprès d'un domaine distant
```



```
monSon.load ( new URLRequest ("http://www.monDomaineDistant.org/son.mp3") );
```

Rappelez-vous, par défaut le lecteur Flash tente de charger le fichier de régulation à la racine du serveur.

Dans le code précédent, le lecteur Flash tentera de charger le fichier de régulation à l'adresse suivante :

```
http://www.monDomaineDistant.org/crossdomain.xml
```

Si nous souhaitons spécifier un emplacement différent, nous utiliserons la méthode `loadPolicyFile` définie par la classe `flash.system.Security`.

## A retenir

- La classe `SoundLoaderContext` permet de préciser la durée de mise en mémoire tampon ainsi qu'une indication concernant le chargement du fichier de régulation.

## Transformation du son

Afin de modifier le volume ou la balance d'un son, nous devons utiliser la classe `SoundTransform`.

Le moyen le plus simple pour modifier le son est d'extraire l'objet de transformation audio par la propriété `soundTransform` de l'objet `SoundChannel` :

```
// instantiation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// récupération de l'objet SoundTransform associé au son en cours de lecture
var transformationSon:SoundTransform = canalSon.soundTransform;
```

Différentes propriétés sont définies par la classe `SoundTransform` dont voici le détail :

- `leftToLeft` : indique la quantité d'entrée gauche à émettre dans le haut-parleur gauche.
- `leftToRight` : indique la quantité d'entrée gauche à émettre dans le haut-parleur droit.
- `pan` : définit la balance du son, la valeur du paramètre varie de -1 à 1.
- `rightToLeft` : indique la quantité d'entrée droite à émettre dans le haut-parleur gauche.
- `rightToRight` : indique la quantité d'entrée droite à émettre dans le haut-parleur droit.

- **volume** : détermine la puissance du volume. Le paramètre varie entre 0 pour un son nul, et 1 pour le volume maximal.

Il est important de noter que la modification du son ne se fait plus par l'intermédiaire de méthodes telles `setVolume` ou `setPan` ou autres.

Pour modifier le volume d'un son, nous devons procéder en plusieurs étapes précises :

1. Créer ou récupérer un objet de transformation `SoundTransform`.
2. Modifier la propriété `volume` de ce dernier.
3. Affecter à nouveau l'objet de transformation à la propriété `soundTransform` de l'objet `SoundChannel`.

Dans le code suivant nous réduisons le volume du son en cours de lecture de 50% :

```
// instantiation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// récupération de l'objet SoundTransform associé au son en cours de lecture
var transformationSon:SoundTransform = canalSon.soundTransform;

// réduction du volume de 50%
transformationSon.volume = .5;

// application de l'objet de transformation
canalSon.soundTransform = transformationSon
```

De la même manière nous pouvons modifier la balance horizontale du son à l'aide de la propriété `pan` :

```
// instantiation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// récupération de l'objet SoundTransform associé au son en cours de lecture
var transformationSon:SoundTransform = canalSon.soundTransform;

// réduction du volume de 50%
transformationSon.volume = .5;

// passage de la totalité du son dans le canal droit
transformationSon.pan = 1;

// application de l'objet de transformation
canalSon.soundTransform = transformationSon;
```

Nous nous rendons compte que la manipulation du son ne s'avère pas simplifiée, il serait intéressant de concevoir une classe appropriée permettant de rendre transparentes toutes ces manipulations.

Pour cela, nous allons concevoir une classe nommée `Amplificateur`. Celle-ci contiendra différentes méthodes telles `affecteVolume`, `affecteBalance`, `recupereVolume` et `recupereBalance`.

Dans un paquetage `org.bytearray.media` nous créons la classe `Amplificateur` suivante :

```
package org.bytearray.media
{
    public class Amplificateur
    {
        public function Amplificateur ()
        {
        }
    }
}
```

Le constructeur de la classe `Amplificateur` nécessite en paramètre le canal audio à modifier. Afin de l'accueillir nous ajoutons un paramètre `pCanal` :

```
package org.bytearray.media
{
    import flash.media.SoundChannel;

    public class Amplificateur
    {
        private var canalSon:SoundChannel;

        public function Amplificateur ( pCanal:SoundChannel )
        {
            canalSon = pCanal;
        }
    }
}
```

Puis nous ajoutons les méthodes spécifiques permettant de modifier le volume ou la balance :

```
package org.bytearray.media
```

```
{  
  
    import flash.media.SoundChannel;  
    import flash.media.SoundTransform;  
  
    public class Amplificateur  
    {  
  
        private var canalSon:SoundChannel;  
        private var transformation:SoundTransform;  
  
        public function Amplificateur ( pCanal:SoundChannel )  
        {  
  
            canalSon = pCanal;  
  
            transformation = pCanal.soundTransform;  
  
        }  
  
        public function affecteVolume ( pVolume:Number ):void  
        {  
  
            transformation.volume = pVolume;  
  
            canalSon.soundTransform = transformation;  
  
        }  
  
        public function recupereVolume ():Number  
        {  
  
            return canalSon.soundTransform.volume;  
  
        }  
  
        public function affecteBalance ( pBalance:Number ):void  
        {  
  
            transformation.pan = pBalance;  
  
            canalSon.soundTransform = transformation;  
  
        }  
  
        public function recupereBalance ():Number  
        {  
  
            return canalSon.soundTransform.pan;  
  
        }  
  
    }  
}
```

La classe `Amplificateur` s'occupe uniquement de la transformation du son. La création de l'objet `Sound` ne lui est pas associée, cela rendrait notre classe rigide.

La modification du volume ou de la balance est ainsi simplifiée, dans le code suivant nous réduisons le volume de 50 % :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;

// instantiation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// création du mixeur de son
var monAmpli:Amplificateur = new Amplificateur( canalSon );

// réduction du volume de 50%
monAmpli.affecteVolume ( .5 );
```

De la même manière, nous pouvons modifier la balance horizontale du son à l'aide de la méthode `affecteBalance` :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;

// instantiation d'un objet Sound, la méthode load est automatiquement appelée
var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// lecture du son
var canalSon:SoundChannel = son.play();

// création du mixeur de son
var monAmpli:Amplificateur = new Amplificateur( canalSon );

// modification de la balance horizontale du son dans le haut parleur droit
monAmpli.affecteBalance ( 1 );
```

Nous allons enrichir la classe `Amplificateur` en ajoutant une nouvelle méthode nommée `appliqueEffet`.

Celle-ci permettra l'ajout d'effets appliqués aux sons. Nous allons revoir quelques notions essentielles propre à la programmation orientée objet dans cet exemple.

L'idée est de pouvoir passer à la méthode `appliqueEffet` un effet spécifique. Le code suivant illustre le concept :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play();
```

```
var monAmpli:Amplificateur = new Amplificateur( canalSon );  
  
// création d'un effet de fondu  
var monEffet:Fondu = new Fondu ();  
  
// application de l'effet  
monAmpli.appliqueEffet ( monEffet );
```

Nous pourrions alors imaginer de signer la méthode `appliqueEffet` en spécifiant un paramètre de type `Fondu` :

```
public function appliqueEffet ( pEffet:Fondu ):void  
{  
}
```

Malheureusement, cette orientation verrait rapidement ses limites, car nous ne pourrions passer en paramètre que des effets de type `Fondu`.

Afin de pouvoir passer n'importe quel type d'effets, nous devons trouver un type commun à tous les effets et typer notre paramètre du même type. Afin de bénéficier d'un type commun, nous pensons immédiatement à la notion d'héritage traitée lors du chapitre 8 intitulé *Programmation orientée objet*.

Nous allons donc créer une classe `EffetSonore` au sein du paquetage `org.bytearray.media.effets` dont toutes les classes d'effets devront hériter.

Celle-ci définit une méthode `executeEffet` que toutes les classes enfants doivent surcharger afin d'implémenter leur propre effet :

```
package org.bytearray.media.effets  
{  
    import flash.media.SoundChannel;  
    public class EffetSonore  
    {  
        public function EffetSonore ()  
        {  
        }  
        public function executeEffet ( pCanal:SoundChannel ):void  
        {  
        }  
    }  
}
```

```
}
```

```
}
```

Puis nous étendons la classe `EffetSonore` à travers la classe `Fondu` tout en surchargeant la méthode `executeEffet` :

```
package org.bytearray.media.effets

{

    import flash.media.SoundChannel;

    public class Fondu extends EffetSonore

    {

        public function Fondu ()

        {

        }

        override public function executeEffet ( pCanal:SoundChannel ):void

        {

            trace("application de l'effet sonore");

        }

    }

}
```

Grâce à cette approche, les classes d'effets devront toujours hériter de la classe `EffetSonore` et posséderont ainsi ce type commun.

Nous pouvons désormais ajouter une méthode `appliqueEffet` à la classe `Amplificateur` en utilisant le type commun `EffetSonore` en paramètre :

```
package org.bytearray.media

{

    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import org.bytearray.media.effets.EffetSonore;

    public class Amplificateur

    {

        private var canalSon:SoundChannel;
        private var transformation:SoundTransform;

        public function Amplificateur ( pCanal:SoundChannel )

        {
```

```
        canalSon = pCanal;

        transformation = pCanal.soundTransform;

    }

    public function affecteVolume ( pVolume:Number ):void
    {

        transformation.volume = pVolume;

        canalSon.soundTransform = transformation;

    }

    public function recupereVolume ():Number
    {

        return canalSon.soundTransform.volume;

    }

    public function affecteBalance ( pBalance:Number ):void
    {

        transformation.pan = pBalance;

        canalSon.soundTransform = transformation;

    }

    public function recupereBalance ():Number
    {

        return canalSon.soundTransform.pan;

    }

    public function appliqueEffet ( pEffet:EffetSonore ):void
    {

        pEffet.executeEffet ( canalSon );

    }

}

}
```

En testant le code suivant :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );
```



```
var canalSon:SoundChannel = son.play();

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu
var monEffet:Fondu = new Fondu ();

// application de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Le message suivant est affiché dans la fenêtre de sortie :

```
application de l'effet sonore
```

Nous retrouvons dans le code précédent les avantages liés à la *liaison dynamique* du *polymorphisme*.

Lors de la compilation, le compilateur ne sait pas quel sera le type exact de l'objet référencé par le paramètre `pEffet`. La définition de la méthode `executeEffet` qui sera déclenchée est évaluée à l'exécution.

La classe `Amplificateur` possède maintenant une nouvelle méthode `appliqueEffet`.

Afin d'achever la classe `Fondu`, il ne nous reste plus qu'à implémenter l'effet au sein de celle-ci :

```
package org.bytearray.media.effets

{

    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import fl.transitions.Tween;
    import fl.transitions.easing.Regular;
    import fl.transitions.TweenEvent;

    public class Fondu extends EffetSonore

    {

        private var duree:Number;
        private var volume:Number;
        private var canalSon:SoundChannel;
        private var transformation:SoundTransform;
        private var objetTween:Tween;

        public function Fondu ( pDuree:Number, pVolume:Number )

        {

            duree = pDuree;

            volume = pVolume;

        }

    }

}
```

```
        override public function executeEffet ( pCanal:SoundChannel ):void
        {

            canalSon = pCanal;

            transformation = canalSon.soundTransform;

            objetTween = new Tween ( transformation, "volume",
Regular.easeInOut, transformation.volume, volume, duree, true );

            objetTween.addEventListener ( TweenEvent.MOTION_CHANGE ,
appliqueEffet );

            objetTween.addEventListener ( TweenEvent.MOTION_FINISH ,
effetTermine );

        }

        private function appliqueEffet ( pEvt:TweenEvent ):void
        {

            canalSon.soundTransform = transformation;

        }

        private function effetTermine ( pEvt:TweenEvent ):void
        {

            objetTween.removeEventListener( TweenEvent.MOTION_CHANGE,
appliqueEffet );

        }

    }
}
```

La classe **Fondu** accepte deux paramètres dont voici le détail :

- **pDuree** : la durée du fondu.
- **pVolume** : le niveau de volume vers lequel le fondu se dirige.

Dans le code suivant, nous appliquons un effet de fondu de 3 secondes vers un volume à 0 :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play();

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu vers un volume à 0 en 3 secondes
var monEffet:Fondu = new Fondu ( 3, 0 );
```

```
// application de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Nous pouvons réduire le volume à 0 puis augmenter progressivement le son jusqu'à un volume de 1 en 5 secondes.

Un objet `SoundTransform` peut être passé en troisième paramètre de la méthode `play` :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet Fondu
import org.bytearray.media.effets.Fondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

// le son est lu avec un volume à 0
var canalSon:SoundChannel = son.play ( 0, 0, new SoundTransform ( 0 ) );

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu vers un volume à 1 en 5 secondes
var monEffet:Fondu = new Fondu ( 5, 1 );

// application de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Nous pouvons ainsi créer d'autres types d'effets. Afin d'étendre le concept d'effets nous allons créer une classe `AutoBalance`. Celle-ci provoquera une balance horizontale entre les deux hauts parleurs.

Voici le code de la classe `AutoBalance` :

```
package org.bytearray.media.effets

{

    import flash.events.TimerEvent;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import flash.utils.Timer;
    import fl.transitions.easing.Regular;

    public class AutoBalance extends EffetSonore

    {

        private var duree:Number;
        private var vitesse:Number;
        private var balance:Number;
        private var i:Number;
        private var canalSon:SoundChannel;
        private var transformation:SoundTransform;
        private var minuteur:Timer;
        private var minuteurArret:Timer;

        public function AutoBalance ( pDuree:Number, pVitesse:Number )

        {
```

```
        balance = i = 0;

        duree = pDuree;
        vitesse = pVitesse;

        minuteur = new Timer ( 100, 0 );
        minuteurArret = new Timer ( pDuree * 1000, 1 );
        minuteur.addEventListener ( TimerEvent.TIMER, appliqueEffet );
        minuteurArret.addEventListener ( TimerEvent.TIMER_COMPLETE,
    effetTermine );
    }

    override public function executeEffet ( pCanal:SoundChannel ):void
    {
        canalSon = pCanal;
        transformation = canalSon.soundTransform;
        minuteur.start();
        minuteurArret.start();
    }

    private function appliqueEffet ( pEvt:TimerEvent ):void
    {
        balance = Math.sin( i += vitesse );
        transformation.pan = balance;
        canalSon.soundTransform = transformation;
    }

    private function effetTermine ( pEvt:TimerEvent ):void
    {
        minuteur.stop();
        transformation.pan = 0;
        canalSon.soundTransform = transformation;
    }
}
```

La classe `AutoBalance` accepte deux paramètres dont voici le détail :

- `pDuree` : la durée de la balance.
- `pVitesse` : la vitesse de la balance horizontale.

La méthode écouteur `appliqueEffet` fait osciller la propriété `balance` entre -1 et 1 grâce à la méthode `sin` de la classe `Math`.

Dans le code suivant, nous appliquons un effet de balance pendant 15 secondes avec une vitesse réduite :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe d'effet AutoBalance
import org.bytearray.media.effets.AutoBalance;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play ();

var monAmpli:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de balance automatique pendant 15 secondes avec une
// vitesse réduite
var monEffet:AutoBalance = new AutoBalance ( 15, .1 );

// application de l'effet
monAmpli.appliqueEffet ( monEffet );
```

Nous avons eu recours à l'héritage afin de bénéficier du *polymorphisme* et d'un type commun, malheureusement notre conception souffre d'une faiblesse importante.

Que se passe-t-il si nous souhaitons ajouter un nouvel effet, héritant déjà d'une classe spécifique ?

Il nous serait impossible d'étendre la classe `EffetSonore`, l'héritage multiple étant impossible en ActionScript 3.

Souvenez-vous, nous avons vu lors du chapitre 8 intitulé *Programmation orientée objet* que l'héritage n'était pas la seule solution afin d'obtenir un ensemble d'objets ayant un type commun.

Il est possible de faire partager à plusieurs classes un même type grâce aux interfaces. Au lieu de définir une classe `EffetSonore` dont toutes les classes d'effets doivent hériter, nous allons simplement créer une interface `IEffetSonore` que toute classe d'effet se devra d'implémenter.

De par l'implémentation, toutes les classes d'effets seront de leurs types respectifs ainsi que du type `IEffetSonore`.

Pour cela, nous définissons l'interface `IEffetSonore` suivante au sein du paquetage `org.bytearray.media.effets` :

```
package org.bytearray.media.effets
{
    import flash.media.SoundChannel;

    public interface IEffetSonore
    {
        function executeEffet ( pCanal:SoundChannel ):void;
    }
}
```

Cette interface définit une seule méthode `executeEffet` que chaque classe d'effet se doit d'implémenter. Souvenez-vous, cette même méthode était surchargée au sein des sous classes dans notre exemple précédent.

Nous allons à présent modifier les classes `Fondu` et `AutoBalance` en implémentant l'interface `IEffetSonore`.

Notez que la méthode `executeEffet` ne doit plus être marquée comme méthode surchargeante, nous supprimons donc l'attribut `override` :

```
package org.bytearray.media.effets
{
    import flash.events.TimerEvent;
    import flash.media.Sound;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import flash.utils.Timer;
    import fl.transitions.Tween;
    import fl.transitions.easing.Regular;
    import fl.transitions.TweenEvent;

    public class Fondu implements IEffetSonore
    {
        private var duree:Number;
        private var volume:Number;
        private var canalSon:SoundChannel;
        private var transformation:SoundTransform;
        private var objetTween:Tween;

        public function Fondu ( pDuree:Number, pVolume:Number )
        {
            duree = pDuree;
            volume = pVolume;
        }
    }
}
```

```
public function executeEffet ( pCanal:SoundChannel ):void
{
    canalSon = pCanal;

    transformation = canalSon.soundTransform;

    objetTween = new Tween ( transformation, "volume",
Regular.easeInOut, transformation.volume, destination, duree, true );

    objetTween.addEventListener ( TweenEvent.MOTION_CHANGE ,
appliqueEffet );

    objetTween.addEventListener ( TweenEvent.MOTION_FINISH ,
effetTermine );
}

private function appliqueEffet ( pEvt:TweenEvent ):void
{
    canalSon.soundTransform = transformation;
}

private function effetTermine ( pEvt:TweenEvent ):void
{
    objetTween.removeEventListener( TweenEvent.MOTION_CHANGE,
appliqueEffet );
}
}
```

La classe `AutoBalance` implémente aussi la classe `IEffetSonore` :

```
package org.bytearray.media.effets
{
    import flash.events.TimerEvent;
    import flash.media.SoundChannel;
    import flash.media.SoundTransform;
    import flash.utils.Timer;
    import fl.transitions.easing.Regular;

    public class AutoBalance implements IEffetSonore
    {
        private var duree:Number;
        private var vitesse:Number;
        private var balance:Number;
        private var i:Number;
        private var canalSon:SoundChannel;
```

```
private var transformation:SoundTransform;
private var minuteur:Timer;
private var minuteurArret:Timer;

public function AutoBalance ( pDuree:Number, pVitesse:Number )
{
    balance = i = 0;

    duree = pDuree;

    vitesse = pVitesse;

    minuteur = new Timer ( 100, 0 );

    minuteurArret = new Timer ( pDuree * 1000, 1 );

    minuteur.addEventListener ( TimerEvent.TIMER, appliqueEffet );
    minuteurArret.addEventListener ( TimerEvent.TIMER_COMPLETE,
    effetTermine );
}

public function executeEffet ( pCanal:SoundChannel ):void
{
    canalSon = pCanal;

    transformation = canalSon.soundTransform;

    minuteur.start();

    minuteurArret.start();
}

private function appliqueEffet ( pEvt:TimerEvent ):void
{
    balance = Math.sin( i += vitesse );

    transformation.pan = balance;

    canalSon.soundTransform = transformation;
}

private function effetTermine ( pEvt:TimerEvent ):void
{
    minuteur.stop();

    transformation.pan = 0;

    canalSon.soundTransform = transformation;
}
```



```
}
```

```
}
```

En implémentant l'interface `IEffetSonore` les classes d'effets sont obligées de définir une méthode `executeEffet`, le cas échéant la compilation est impossible le message suivant est affiché :

```
1044: La méthode d'interface executeEffet de l'espace de nom  
org.bytearray.media.effets:IEffetSonore n'est pas implémentée par la classe  
org.bytearray.media.effets:Fondu.
```

La méthode `appliqueEffet` de la classe `Amplificateur` accepte désormais un paramètre de type `IEffetSonore` :

```
package org.bytearray.media  
  
{  
  
    import flash.errors.IllegalOperationError;  
    import flash.media.SoundChannel;  
    import flash.media.SoundTransform;  
    import org.bytearray.media.effets.IEffetSonore;  
  
    public class Amplificateur  
    {  
  
        private var canalSon:SoundChannel;  
        private var transformation:SoundTransform;  
  
        public function Amplificateur ( pCanal:SoundChannel )  
        {  
  
            canalSon = pCanal;  
  
            transformation = pCanal.soundTransform;  
  
        }  
  
        public function affecteVolume ( pVolume:Number ):void  
        {  
  
            transformation.volume = pVolume;  
  
            canalSon.soundTransform = transformation;  
  
        }  
  
        public function recupereVolume ():Number  
        {  
  
            return canalSon.soundTransform.volume;  
  
        }  
  
        public function affecteBalance ( pBalance:Number ):void
```

```
        {  
            transformation.pan = pBalance;  
            canalSon.soundTransform = transformation;  
        }  
        public function recupereBalance ():Number  
        {  
            return canalSon.soundTransform.pan;  
        }  
        public function appliqueEffet ( pEffet:IEffetSonore ):void  
        {  
            pEffet.executeEffet ( canalSon );  
        }  
    }  
}
```

Grâce à la notion d'interfaces, les classes `Fondu` et `AutoBalance` sont de type `IEffetSonore`. Si une sous-classe souhaite devenir une classe d'effet, celle-ci n'a qu'à implémenter l'interface `IEffetSonore` et définir la méthode `executeEffet`.

Pour terminer, nous allons ajouter la diffusion d'un événement depuis la classe `Fondu` afin de faciliter son utilisation.

Celle-ci diffusera les deux événements suivants :

- `EvenementFondu.DEMARRE` : l'effet est démarré.
- `EvenementFondu.TRANSITION` : l'effet est en cours.
- `EvenementFondu.TERMINE` : l'effet est terminé.

Afin de pouvoir diffuser ces derniers, la classe `Fondu` étend la classe `EventDispatcher` :

```
| public class Fondu extends EventDispatcher implements IEffectSonore
```

Veillez à importer la classe `EventDispatcher` :

```
| import flash.events.EventDispatcher;
```

Puis nous définissons la classe `EvenementFondu` au sein du paquetage `org.bytearray.media.effets.events` :

```
| package org.bytearray.media.effets.events
```

```
{  
  
    import flash.events.Event;  
  
    public class EvenementFondu extends Event  
    {  
  
        public static const DEMARRE:String = "demarre";  
        public static const TRANSITION:String = "transition";  
        public static const TERMINE:String = "termine";  
  
        public function EvenementFondu ( pType:String )  
        {  
  
            super( pType, false, false );  
  
        }  
  
        public override function clone ():Event  
        {  
  
            return new EvenementFondu ( type );  
  
        }  
  
        public override function toString ():String  
        {  
  
            return '[EvenementFondu type="'+ type +' " bubbles=' + bubbles + '  
eventPhase='+ eventPhase + ' cancelable=' + cancelable + ']';  
  
        }  
  
    }  
  
}
```

Puis nous diffusons les événements appropriés pour chaque phase liée à l'effet :

```
package org.bytearray.media.effets  
  
{  
  
    import flash.events.Event;  
    import flash.events.EventDispatcher;  
    import flash.events.TimerEvent;  
    import flash.media.Sound;  
    import flash.media.SoundChannel;  
    import flash.media.SoundTransform;  
    import flash.utils.Timer;  
    import fl.transitions.Tween;  
    import fl.transitions.easing.Regular;  
    import fl.transitions.TweenEvent;  
    import org.bytearray.media.effets.evenements.EvenementFondu;  
  
    public class Fondu extends EventDispatcher implements IEffetSonore
```

```
{

    private var duree:Number;
    private var destination:Number;
    private var canalSon:SoundChannel;
    private var transformation:SoundTransform;
    private var objetTween:Tween;

    public function Fondu ( pDuree:Number, pDestination:Number )

    {

        duree = pDuree;

        destination = pDestination;

    }

    public function executeEffet ( pCanal:SoundChannel ):void

    {

        canalSon = pCanal;

        transformation = canalSon.soundTransform;

        objetTween = new Tween ( transformation, "volume",
Regular.easeInOut, transformation.volume, destination, duree, true );

        dispatchEvent ( new EvenementFondu (EvenementFondu.DEMARRE) );

        objetTween.addEventListener ( TweenEvent.MOTION_CHANGE ,
appliqueEffet );

        objetTween.addEventListener ( TweenEvent.MOTION_FINISH ,
effetTermine );

    }

    private function appliqueEffet ( pEvt:TweenEvent ):void

    {

        canalSon.soundTransform = transformation;

        dispatchEvent ( new EvenementFondu (EvenementFondu.TRANSITION) );

    }

    private function effetTermine ( pEvt:TweenEvent ):void

    {

        objetTween.removeEventListener( TweenEvent.MOTION_CHANGE,
appliqueEffet );

        dispatchEvent ( new EvenementFondu (EvenementFondu.TERMINE) );

    }

}
```

```
}
```

Chaque événement peut ensuite être écouté afin de pouvoir facilement synchroniser l'application :

```
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import des classes d'effets
import org.bytearray.media.effets.AutoBalance;
import org.bytearray.media.effets.Fondu;

// import de la classe EvenementFondu
import org.bytearray.media.effets.evenements.EvenementFondu;

var son:Sound = new Sound ( new URLRequest ("son.mp3") );

var canalSon:SoundChannel = son.play ();

var monMixer:Amplificateur = new Amplificateur( canalSon );

// création d'un effet de fondu vers un volume à 0 en 10 secondes
var monEffet:Fondu = new Fondu ( 10, 0 );

// écoute des événements EvenementFondu.DEMARRE et EvenementFondu.TERMEINE
monEffet.addEventListener ( EvenementFondu.DEMARRE, effetDemarre );
monEffet.addEventListener ( EvenementFondu.TRANSITION, effetEnCours );
monEffet.addEventListener ( EvenementFondu.TERMEINE, effetTermine );

// application de l'effet
monMixer.appliqueEffet ( monEffet );

function effetDemarre ( pEvt:EvenementFondu ):void
{
    trace("effet démarré ");
}

function effetEnCours ( pEvt:EvenementFondu ):void
{
    trace("effet en cours ");
}

function effetTermine ( pEvt:EvenementFondu ):void
{
    trace("effet terminé ");
}
```

A vous d'intégrer les mêmes événements au sein de la classe `AutoBalance` et pourquoi pas d'ajouter de nouvelles classes d'effets !

## A retenir

- Afin de modifier le son plus facilement, nous avons créé une classe `Amplificateur`. Celle-ci possède une méthode `appliqueEffet` permettant d'affecter différents effets sonore.
- Les classes `Fondu` et `AutoBalance` peuvent être utilisées comme effets sonore. D'autres effets peuvent être ajoutés très simplement.
- Les classes d'effets doivent obligatoirement implémenter l'interface `IEffetSonore` afin d'être compatible.
- Après une première approche basée sur l'héritage, nous avons préféré utiliser une interface `IEffetSonore` afin de rendre plus souple la création de nouveaux effets.

## Modification globale du son

Nous venons d'étudier la modification de chaque son de manière individuelle, ActionScript 3 introduit une classe `SoundMixer` permettant de travailler de manière globale sur les sons d'une application.

Celle-ci définit une méthode `stopAll` permettant l'arrêt de tous les sons en cours de lecture :

```
// stoppe tous les sons en cours de lecture  
SoundMixer.stopAll();
```

Si nous souhaitons modifier le volume global, nous pouvons utiliser la propriété statique `soundTransform` de la même classe.

Dans le code suivant, nous réduisons le volume global à 10% :

```
// réduit tous les sons en cours de lecture  
SoundMixer.soundTransform = new SoundTransform ( .1 );
```

Nous allons voir dans la partie suivante, que la classe `SoundMixer` ne se limite pas à cela. Celle-ci nous réserve une fonctionnalité fort intéressante ouvrant de nouvelles possibilités en matière d'application audio.

## A retenir

- La modification d'un son est assurée par la classe `SoundTransform`.
- Un objet `SoundChannel` est renvoyé par la méthode `play` de l'objet `Sound`.
- Les objets de type `SoundChannel` possèdent une propriété `soundTransform` renvoyant un objet de type `SoundTransform`.
- La classe `SoundMixer` permet de travailler de manière globale sur les sons d'une application.

## Lire le spectre d'un son

ActionScript 3 intègre une nouvelle fonctionnalité très intéressante au travers de la méthode `computeSpectrum` de la classe `SoundMixer`.

Celle ci permet de récupérer le spectre de la totalité des sons en cours de lecture afin d'en offrir une représentation graphique.

Attention, le son issu de la classe `Microphone` ne peut être redirigé vers la classe `Sound` et n'est donc pas pris en considération par la méthode `computeSpectrum`.

Dans le cas contraire, cela nous aurait permis de travailler sur la reconnaissance vocale au sein de Flash. Nous pouvons espérer que cette fonctionnalité soit intégrée dans une prochaine version du lecteur.

En attendant, revenons à la méthode `computeSpectrum` dont voici la signature :

```
public static function computeSpectrum(outputArray:ByteArray, FFTMode:Boolean = false, stretchFactor:int = 0):void
```

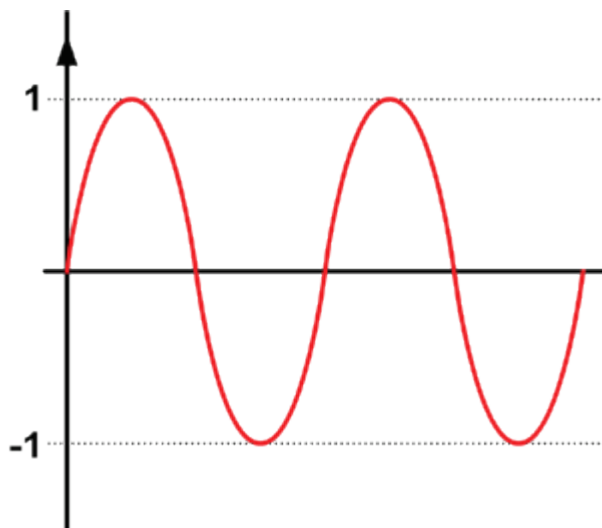
Voici le détail de chacun des paramètres :

- `outputArray` : le tableau binaire dans lequel placer les données liées au spectre du son. Une instance de la classe `ByteArray` est nécessaire.
- `FFTMode` : permet de spécifier si une transformation de Fourier doit être appliquée au spectre généré. Nous verrons plus loin en quoi consiste cette transformation.
- `stretchFactor` : échantillonnage des données générées. La valeur par défaut est 0 c'est-à-dire 44,1 KHz.

Lorsque la méthode `computeSpectrum` est exécutée, celle-ci place au sein du tableau binaire le spectre de la totalité des sons en cours de lecture.

Par défaut, ce spectre est représenté par 512 valeurs oscillant entre -1 et 1, dont les 256 premières valeurs concernent le haut parleur gauche, et les 256 suivantes le haut parleur droit.

La figure 17-2 illustre l'oscillation des valeurs retournées par la méthode `computeSpectrum` :



*Figure 17-2. Oscillation des valeurs retournées par la méthode `computeSpectrum`.*

Afin de garantir un rafraîchissement des données au sein du tableau binaire, nous pouvons placer l'appel de la méthode

`computeSpectrum` au sein d'un événement `Event.ENTER_FRAME` :

```
// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création d'un tableau binaire vide pour accueillir le flux audio
var fluxSpectre:ByteArray = new ByteArray();

// calcul du spectre
addEventListener ( Event.ENTER_FRAME, calculSpectre );

function calculSpectre ( pEvt:Event ):void
{
    // calcul du spectre en continu
    SoundMixer.computeSpectrum( fluxSpectre );

    // affiche : 2048
    trace( fluxSpectre.length );
}
```



Notez que l'utilisation d'un objet `Timer` serait aussi envisageable si nous ne souhaitons pas être lié à la cadence de l'animation.

Nous venons de voir que la méthode `computeSpectrum` renvoie 512 valeurs. Pourtant, lorsque nous accédons à la propriété `length` du tableau `fluxSpectre`, celle-ci nous renvoie 2048.

Comment expliquons-nous cela ?

Dans le cas de l'utilisation de la méthode `computeSpectrum`, les valeurs placées au sein du tableau `fluxSpectre` sont stockées sous la forme de nombres à virgule flottante 32 bits (IEEE 754) codés sur 4 octets.

Chaque index d'un tableau binaire représentant un octet, le stockage d'un flottant requiert 4 octets, donc 4 index. Si nous multiplions les 512 valeurs par 4 nous obtenons bien 2048.

Nous allons concevoir une classe `Equaliseur` afin de représenter graphiquement le spectre.

Au sein du paquetage `org.bytearray.media.spectres` nous définissons la classe `Equaliseur` suivante :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;

    public class Equaliseur extends Bitmap

    {

        public function Equaliseur ()

        {

        }

    }

}
```

Afin d'assurer des performances optimales nous évitons la manipulation de données vectorielles et privilégions l'utilisation de données bitmap.

De ce fait, la classe `Equaliseur` étend la classe `Bitmap`.

Souvenez-vous, nous avons vu au cours du chapitre 12 intitulé *Programmation Bitmap* qu'il était préférable d'utiliser des données bitmap lorsque cela était possible afin d'accélérer la vitesse de rendu.

Nous ajoutons à présent notre mécanisme d'activation et de désactivation de l'objet graphique :

```
package org.bytearray.media.spectres
{
    import flash.display.Bitmap;
    import flash.events.Event;

    public class Equaliseur extends Bitmap
    {
        public function Equaliseur ()
        {
            addEventListener ( Event.ADDED_TO_STAGE, activation );
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
        }

        private function activation ( pEvt:Event ):void
        {
            trace("activation");
        }

        private function desactivation ( pEvt:Event ):void
        {
            trace("desactivation");
        }
    }
}
```

Nous en profitons pour ajouter trois paramètres au constructeur permettant de spécifier les dimensions du spectre ainsi que sa couleur :

```
package org.bytearray.media.spectres
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;

    public class Equaliseur extends Bitmap
    {
        private var largeur:int;
        private var hauteur:int;
        private var couleur:Number;
```

```
public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number )
{
    largeur = pLargeur;
    hauteur = pHauteur;
    couleur = pCouleurSpectre;

    addEventListener ( Event.ADDED_TO_STAGE, activation );
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
}

private function activation ( pEvt:Event ):void
{
    bitmapData = new BitmapData ( largeur, hauteur, false, 0 );
}

private function desactivation ( pEvt:Event ):void
{
    bitmapData.dispose();
}
}
```

Notez que le paramètre `pCouleurSpectre` n'est pas lié à l'instance de `BitmapData` créé. Nous utiliserons la couleur passée en paramètre pour teinter les pixels dessinés plus tard au sein du bitmap.

Afin de dessiner le spectre nous devons d'abord définir la surface à peindre. Au sein de la méthode `activation`, nous affectons à la propriété `bitmapData` héritée une instance de la classe `flash.display.BitmapData`.

Lorsqu'il est supprimé de la liste d'affichage, l'équaliseur est automatiquement désactivé grâce à la méthode `dispose`.

Nous pouvons tester la classe en cours, à l'aide du code suivant :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;

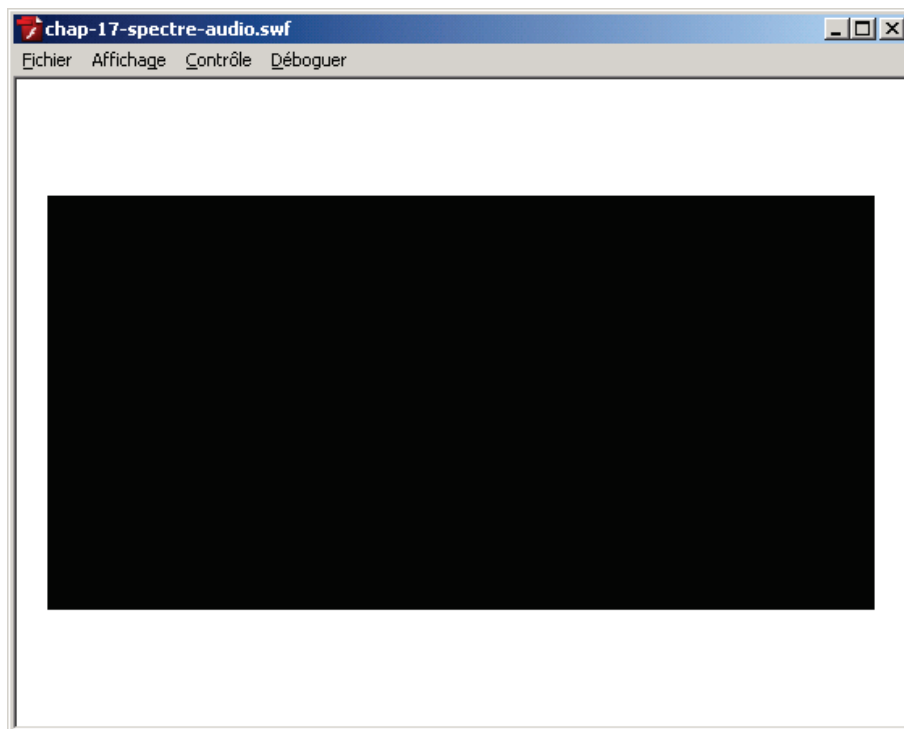
// création d'un égaliseur de 512 pixels de largeur et 256 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 256, 0 );

addChild ( monEqualiseur );

// centrage de l'équaliseur
// >> 1 permet de diviser par 2 de manière plus optimisée ;)
```

```
monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

La figure 17-3 illustre le résultat :



*Figure 17-3. Instance de la classe `Equaliseur`.*

Nous intégrons au sein de la méthode `activation` l'écoute de l'événement `Event.ENTER_FRAME` afin de calculer le spectre :

```
package org.bytearray.media.spectres
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;

    public class Equaliseur extends Bitmap
    {
        private var largeur:int;
        private var hauteur:int;
        private var couleur:Number;
        private var fluxSpectre:ByteArray;

        public function Equaliseur ( pLargeur:Number, pHauteur:Number,
        pCouleurSpectre:Number )
        {
```

```
        largeur = pLargeur;
        hauteur = pHauteur;
        couleur = pCouleurSpectre;

        fluxSpectre = new ByteArray();

        addEventListener ( Event.ADDED_TO_STAGE, activation );
        addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
    }

    private function activation ( pEvt:Event ):void
    {
        bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

        addEventListener ( Event.ENTER_FRAME, calculSpectre );
    }

    private function desactivation ( pEvt:Event ):void
    {
        bitmapData.dispose();
    }

    private function calculSpectre ( pEvt:Event ):void
    {
        SoundMixer.computeSpectrum( fluxSpectre );

        // affiche : 2048
        trace( fluxSpectre.length );
    }
}
}
```

Nous allons nous attarder sur la méthode `calculSpectre` et lire les données du spectre stockées au sein du tableau binaire `fluxSpectre`.

Nous ajoutons la définition de deux propriétés `i` et `oscillation` :

```
private var i:int;
private var oscillation:Number;
```

Puis nous modifions la méthode `calculSpectre` :

```
private function calculSpectre ( pEvt:Event ):void
{
    SoundMixer.computeSpectrum( fluxSpectre );

    i = 512;
```

```
while ( i-- )
{
    oscillation = fluxSpectre.readFloat();

    // affiche : valeur comprise entre -1 et 1
    trace (oscillation);
}
}
```

La boucle `while` intégrée à la méthode `calculSpectre` nous permet de lire les données contenues au sein du tableau `fluxSpectre`.

Contrairement aux tableaux traditionnels, les tableaux binaires possèdent de nombreuses méthodes facilitant leur lecture.

Afin de lire un nombre à virgule flottante, nous devons utiliser la méthode `readFloat` définie par la classe `ByteArray`.

Celle-ci déplace automatiquement la propriété `position` du tableau binaire de 4 index à chaque appel. Les 512 itérations de la boucle permettent donc le parcours des 512 valeurs.

Comme nous l'avons vu précédemment, le spectre est décrit par 512 valeurs oscillant entre -1 et 1. Ces valeurs ne sont pas exploitables graphiquement car trop réduites, nous devons donc les multiplier par une valeur spécifique afin d'obtenir une amplitude suffisante.

Dans le code suivant, nous multiplions les valeurs par la hauteur du spectre divisée par 2 :

```
private function calculSpectre ( pEvt:Event ):void
{
    SoundMixer.computeSpectrum( fluxSpectre );

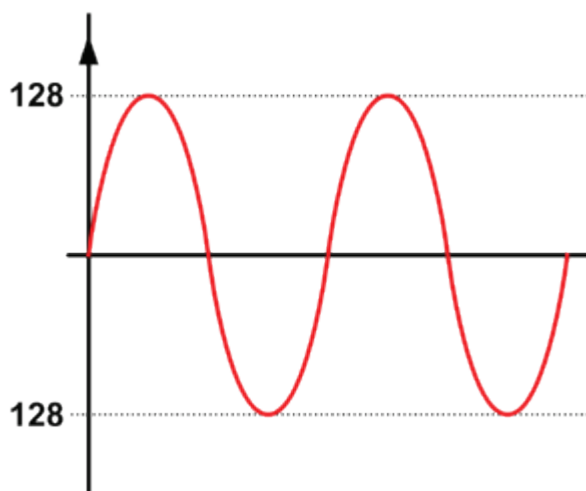
    i = 512;

    while ( i-- )
    {
        oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

        // si la hauteur du spectre est de 256 pixels
        // affiche : valeur comprise entre -128 et 128
        trace (oscillation);
    }
}
```

La propriété `oscillation` évolue désormais entre -128 et 128, cela nous permet de dessiner les bâtonnets constituant notre futur spectre.

La figure 17-4 illustre la nouvelle oscillation pour un spectre d'une hauteur de 256 pixels :



*Figure 17-4. Oscillation des valeurs retournées par la méthode `computeSpectrum`.*

A l'aide de la méthode `fillRect` de l'objet `BitmapData`, nous allons dessiner une succession de bâtonnets représentant le spectre.

Un objet `Rectangle` est utilisé afin de définir la surface de chaque bâtonnet. Celui-ci aura une hauteur définie par la propriété `oscillation`.

Nous définissons une propriété `surface` afin de stocker l'objet `Rectangle` :

```
package org.bytearray.media.spectres
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.geom.Rectangle;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;

    public class Equaliseur extends Bitmap
    {
        private var largeur:int;
        private var hauteur:int;
        private var couleur:Number;
        private var fluxSpectre:ByteArray;
```

```
private var i:int;
private var oscillation:Number;
private var surface:Rectangle;

public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number )

{

    largeur = pLargeur;
    hauteur = pHauteur;
    couleur = pCouleurSpectre;

    surface = new Rectangle ( 0, 0, 3, 4 );

    fluxSpectre = new ByteArray();

    addEventListener ( Event.ADDED_TO_STAGE, activation );
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

}

private function activation ( pEvt:Event ):void

{

    bitmapData = new BitmapData ( largeur, hauteur, false, 0 );
    addEventListener ( Event.ENTER_FRAME, calculSpectre );

}

private function desactivation ( pEvt:Event ):void

{

    bitmapData.dispose();

}

private function calculSpectre ( pEvt:Event ):void

{

    SoundMixer.computeSpectrum( fluxSpectre );

    i = 512;
    while ( i-- )

    {

        oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

        surface.x = i * 4;

        if ( oscillation > 0 )

        {

            surface.y = (bitmapData.height >> 1) - oscillation;
            surface.height = oscillation;


```



```
        } else
        {
            surface.y = (bitmapData.height >> 1);
            surface.height = -oscillation;
        }

        bitmapData.fillRect ( surface, 0xFFFFFFFF );
    }
}
}
```

Nous créons un objet `Rectangle` de 3 pixels de large, cette surface va nous permettre de dessiner chaque bâtonnet de l'équaliseur.

Au sein de la boucle `while` nous déplaçons l'objet `Rectangle` afin de dessiner un bâtonnet tous les 4 pixels.

Rappelez-vous que nous devons lire 512 valeurs et que celles-ci doivent être rendues à l'affichage. Une largeur de 512 pixels minimum est requise afin de pouvoir dessiner la totalité du spectre.

Si nous souhaitons dessiner un spectre de taille réduite, nous devons sauter certaines valeurs du tableau binaire.

Nous devons donc tout d'abord diviser la largeur du spectre spécifiée par 4 afin de déterminer le nombre d'itérations nécessaire pour afficher la totalité des bâtonnets :

```
private function calculSpectre ( pEvt:Event ):void
{
    SoundMixer.computeSpectrum( fluxSpectre );

    i = bitmapData.width / 4;

    while ( i-- )
    {
        oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

        surface.x = i * 4;

        if ( oscillation > 0 )
        {
            surface.y = (bitmapData.height >> 1) - oscillation;
        }
    }
}
```

```
        surface.height = oscillation;
    } else
    {
        surface.y = (bitmapData.height >> 1);
        surface.height = -oscillation;
    }

    bitmapData.fillRect ( surface, 0xFFFFFFFF );
}
}
```

Grâce au code précédent, nous positionnons les bâtonnets sur la largeur du spectre spécifiée, mais nous ne parcourons plus les données totales du tableau binaire `spectreFlux`.

Souvenez-vous que 512 appels à la méthode `readFloat` sont nécessaires pour parcourir le tableau complet, nous devons donc nous arranger pour sauter certaines valeurs tout en s'assurant que nous sommes bien allés jusqu'à la fin du tableau `fluxSpectre`.

Pour cela, nous définissons une propriété `decalage` :

```
|private var decalage:int;
```

Puis nous modifions la méthode `calculSpectre` en sautant certaines valeurs à l'aide de la propriété `position` de l'objet `ByteArray` :

```
private function calculSpectre ( pEvt:Event ):void
{
    SoundMixer.computeSpectrum( fluxSpectre );

    i = bitmapData.width / 4;

    decalage = 2048 / i;

    while ( i-- )
    {
        fluxSpectre.position = i * decalage;

        oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

        surface.x = i * 4;

        if (oscillation > 0 )
        {
            surface.y = (bitmapData.height >> 1) - oscillation;
            surface.height = oscillation;
        }
    }
}
```

```
    } else
    {
        surface.y = (bitmapData.height >> 1);
        surface.height = -oscillation;
    }

    bitmapData.fillRect ( surface, 0xFFFFFFFF );
}
}
```

Afin de comprendre le code précédent, considérons le scénario suivant :

Une largeur de 256 pixels est spécifiée dans le constructeur de la classe `Equaliseur`. En divisant 256 par 4 nous obtenons 64 itérations afin de positionner les bâtonnets représentant le spectre.

Nous divisons 2048 par 64 et obtenons un décalage de 32 octets. Ainsi, pour chaque itération, nous sautons au sein du tableau 32 octets soit 32 index. En fin de boucle nous avons parcouru la totalité du tableau binaire car  $64 * 32 = 2048$ .

En testant la classe `Equaliseur` à l'aide du code suivant :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

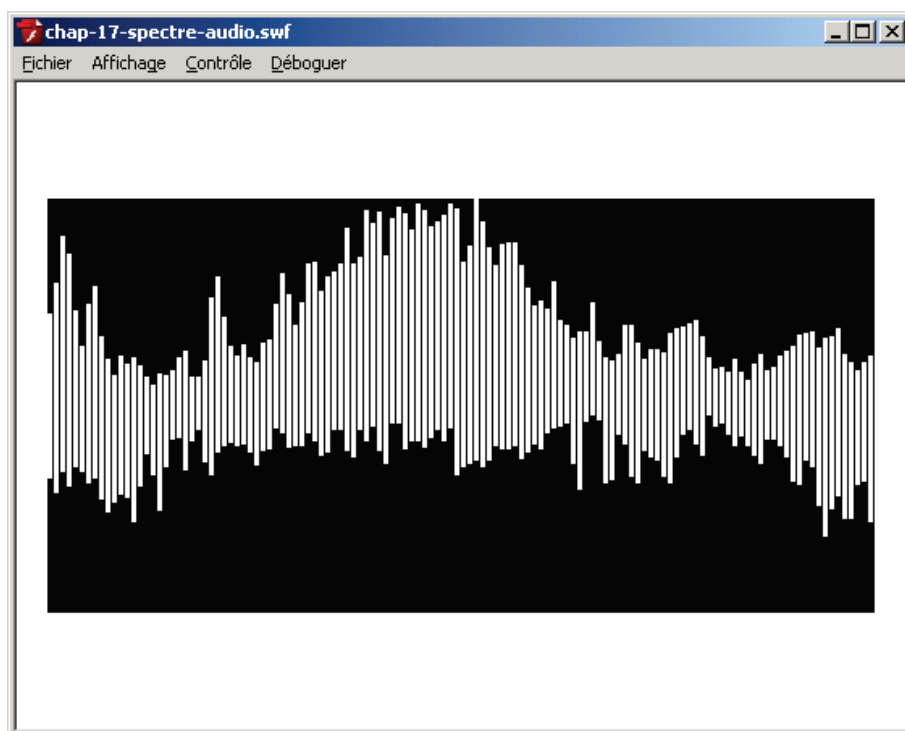
// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création d'un égaliseur de 512 pixels de largeur et 300 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 300, 0 );

addChild ( monEqualiseur );

monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

Nous obtenons le résultat illustré en figure figure 17-5 :



*Figure 17-5. Instance de la classe `Equaliseur`.*

En observant notre égaliseur évoluer, nous remarquons que le spectre dessiné demeure à l’affichage.

Afin de corriger cela, nous ajoutons un nouvel appel à la méthode `fillRect` au sein de la méthode `calculSpectre` :

```
private function calculSpectre ( pEvt:Event ):void
{
    SoundMixer.computeSpectrum( fluxSpectre );

    bitmapData.fillRect ( bitmapData.rect, 0 );

    i = bitmapData.width / 4;
    decalage = 2048 / i;
    while ( i-- )
    {
        fluxSpectre.position = i * decalage;
        oscillation = fluxSpectre.readFloat() * (hauteur >> 1);
        surface.x = i * 4;
        if ( oscillation > 0 )
        {
```

```
        surface.y = (bitmapData.height >> 1) - oscillation;
        surface.height = oscillation;

    } else

    {

        surface.y = (bitmapData.height >> 1);
        surface.height = -oscillation;

    }

    bitmapData.fillRect ( surface, 0xFFFFFFFF );

}

}
```

Le premier appel à la méthode `fillRect` permet de supprimer les pixels précédents. En testant à nouveau notre égaliseur, nous remarquons que les bâtonnets disparaissent à présent, l'égaliseur est correctement rafraîchi.

Nous allons modifier le rendu du spectre en ajoutant une dissolution des pixels progressive afin de donner un effet de fondu plus esthétique. Pour dissoudre les pixels, nous utilisons un filtre de flou appliqué en continu.

Pour cela, nous utilisons la méthode `applyFilter` de la classe `BitmapData` :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;

    public class Equaliseur extends Bitmap
    {

        private var largeur:int;
        private var hauteur:int;
        private var couleur:Number;
        private var fluxSpectre:ByteArray;
        private var i:int;
        private var oscillation:Number;
        private var surface:Rectangle;
        private var decalage:int;
        private var filtreFlou:BlurFilter;
        private var point:Point;
```

```
        public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number )

        {

            largeur = pLargeur;
            hauteur = pHauteur;
            couleur = pCouleurSpectre;

            surface = new Rectangle ( 0, 0, 3, 4 );

            fluxSpectre = new ByteArray();

            filtreFlou = new BlurFilter ( 0, 4, 4 );

            point = new Point();

            addEventListener ( Event.ADDED_TO_STAGE, activation );
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

        }

        private function activation ( pEvt:Event ):void

        {

            bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

            addEventListener ( Event.ENTER_FRAME, calculSpectre );

        }

        private function desactivation ( pEvt:Event ):void

        {

            bitmapData.dispose();

        }

        private function calculSpectre ( pEvt:Event ):void

        {

            SoundMixer.computeSpectrum( fluxSpectre );

            i = bitmapData.width / 4;

            decalage = 2048 / i;

            while ( i-- )

            {

                fluxSpectre.position = i * decalage;

                oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

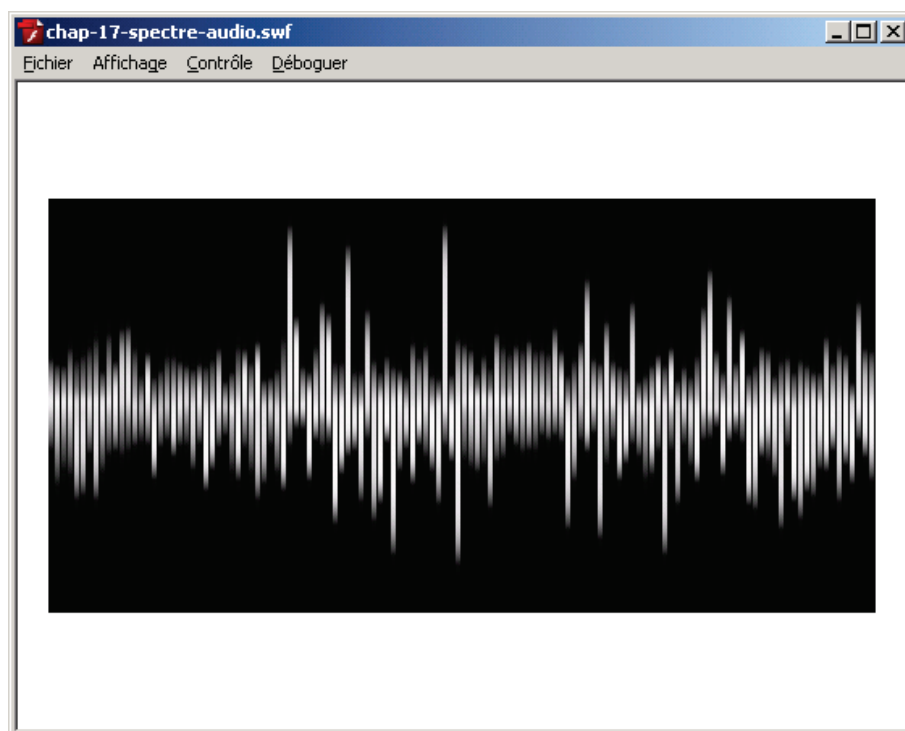
                surface.x = i * 4;

                if ( oscillation > 0 )
```

```
        {  
            surface.y = (bitmapData.height >> 1) - oscillation;  
            surface.height = oscillation;  
        } else  
        {  
            surface.y = (bitmapData.height >> 1);  
            surface.height = -oscillation;  
        }  
  
        bitmapData.fillRect ( surface, 0xFFFFFFFF );  
    }  
  
    bitmapData.applyFilter ( bitmapData, bitmapData.rect, point,  
filtreFlou );  
}  
}
```

Nous passons en paramètre à la méthode `applyFilter` l'objet `BitmapData` en cours, ainsi que sa propriété `rect` afin de définir la surface sur laquelle appliquer le filtre. L'objet `Point` passé en dernier paramètre permet d'indiquer le point de départ d'affectation du filtre.

La figure 17-6 illustre le résultat :



*Figure 17-6. Equaliseur avec fondu progressif.*

Notre égaliseur commence à prendre forme, il ne nous reste plus qu'à ajouter une couleur de spectre aléatoire.

Pour cela nous importons la classe `BitmapOutils` du paquetage `org.bytearray.ouutils` développée au cours du chapitre 12 intitulé *Programmation bitmap*, puis nous créons un objet `ColorTransform` appliqué en continu au spectre :

```
package org.bytearray.media.spectres
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.geom.ColorTransform;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;
    import org.bytearray.ouutils.BitmapOutils;

    public class Equaliseur extends Bitmap
    {
        private var largeur:int;
        private var hauteur:int;
        private var couleur:Number;
```



```

private var fluxSpectre:ByteArray;
private var i:int;
private var oscillation:Number;
private var surface:Rectangle;
private var decalage:int;
private var filtreFlou:BlurFilter;
private var point:Point;
private var transformationCouleur:ColorTransform;

public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number )

{

    largeur = pLargeur;
    hauteur = pHauteur;
    couleur = pCouleurSpectre;

    surface = new Rectangle ( 0, 0, 3, 4 );

    fluxSpectre = new ByteArray();

    filtreFlou = new BlurFilter ( 0, 4, 4 );

    point = new Point();

    var composants:Object = BitmapUtils.hexRgb ( pCouleurSpectre );

    transformationCouleur = new ColorTransform (
composants.rouge/255, composants.vert/255, composants.bleu/255 );

    addEventListener ( Event.ADDED_TO_STAGE, activation );
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

}

private function activation ( pEvt:Event ):void

{

    bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

    addEventListener ( Event.ENTER_FRAME, calculSpectre );

}

private function desactivation ( pEvt:Event ):void

{

    bitmapData.dispose();

}

private function calculSpectre ( pEvt:Event ):void

{

    SoundMixer.computeSpectrum( fluxSpectre );

    i = bitmapData.width / 4;

```

```
        decalage = 2048 / i;

        while ( i-- )
        {
            fluxSpectre.position = i * decalage;

            oscillation = fluxSpectre.readFloat() * (hauteur >> 1);

            surface.x = i * 4;

            if ( oscillation > 0 )
            {
                surface.y = (bitmapData.height >> 1) - oscillation;
                surface.height = oscillation;
            } else
            {
                surface.y = (bitmapData.height >> 1);
                surface.height = -oscillation;
            }

            bitmapData.fillRect ( surface, 0xFFFFFFFF );
        }

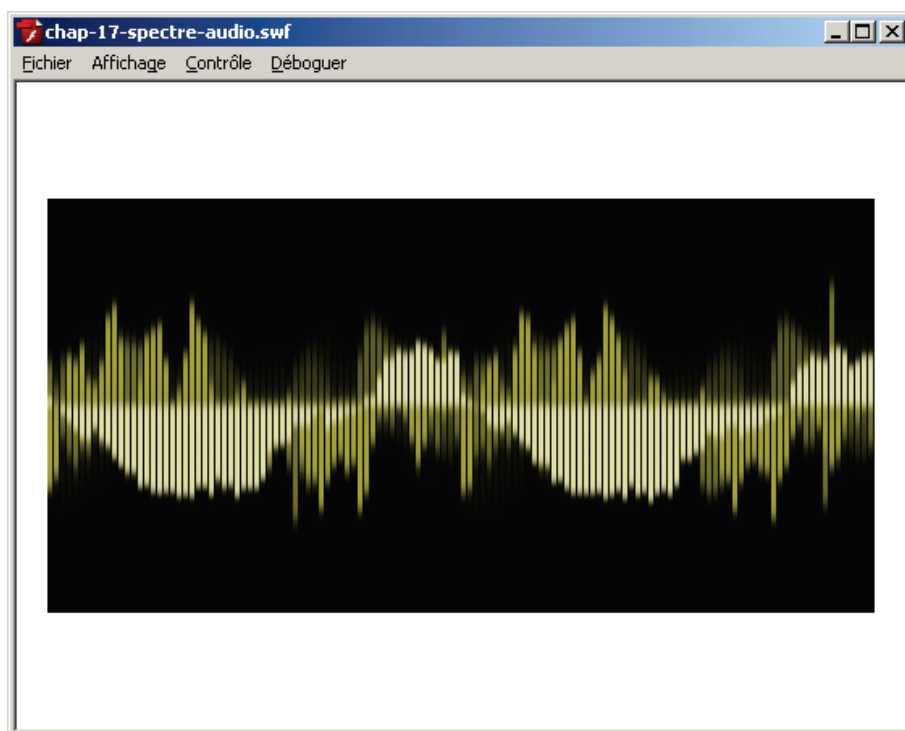
        bitmapData.applyFilter ( bitmapData, bitmapData.rect, point,
filtreFlou );

        bitmapData.colorTransform ( bitmapData.rect,
transformationCouleur );
    }
}
```

Puis nous passons la couleur spécifique lors de l’instanciation de l’objet **Equaliseur** :

```
// création d'un égaliseur de 512 pixels de largeur et 300 pixels de hauteur
de couleur jaune
var monEqualiseur:Equaliseur = new Equaliseur( 512, 300, 0xDDDDA5 );
```

La figure 17-7 illustre le résultat :



*Figure 17-7. Equaliseur audio en couleurs.*

Nous pouvons modifier les dimensions de l'équaliseur de manière significative afin de le réduire à un élément secondaire :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

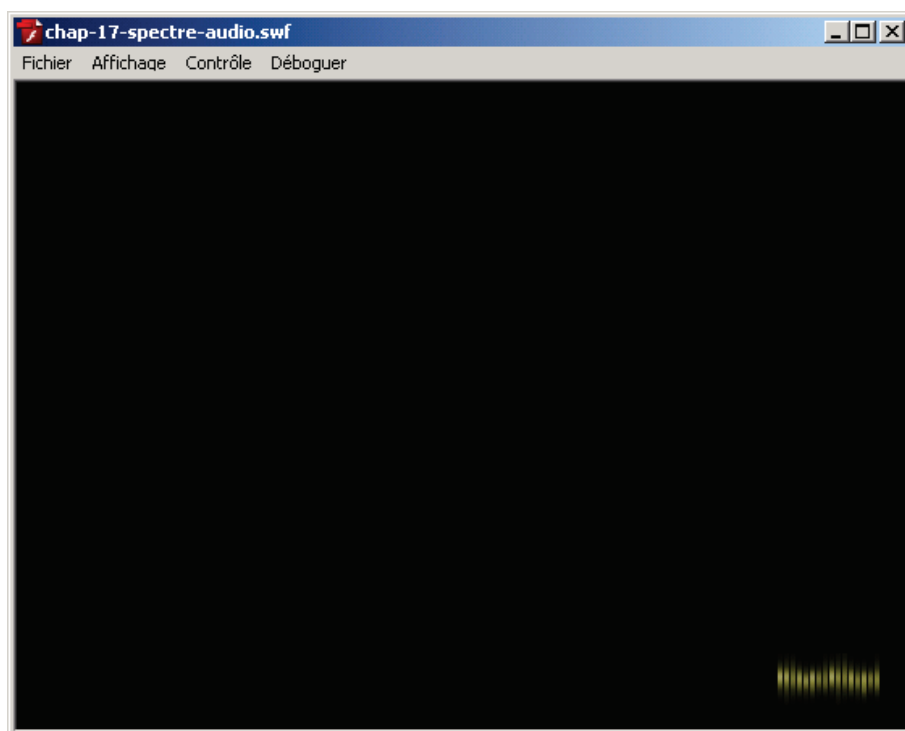
// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création d'un égaliseur de 64 pixels de largeur et 35 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 64, 35, 0xDDDDA5 );

addChild ( monEqualiseur );

// placement de l'équaliseur
monEqualiseur.x = stage.stageWidth - monEqualiseur.width - 15;
monEqualiseur.y = stage.stageHeight - monEqualiseur.height - 15;
```

La figure 17-8 illustre le résultat :



*Figure 17-8. Spectre audio réduit.*

Nous pouvons tester à présent la classe `Amplificateur` développée précédemment afin de voir le spectre modifié en temps réel.

Nous importons la classe `Amplificateur` puis nous appliquons un effet de balance en plaçant la totalité du son dans le haut parleur gauche :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création de l'objet Amplificateur
var monAmplificateur:Amplificateur = new Amplificateur ( canalSon );

// balance horizontale du son dans le haut parleur gauche
monAmplificateur.affecteBalance ( -1 );

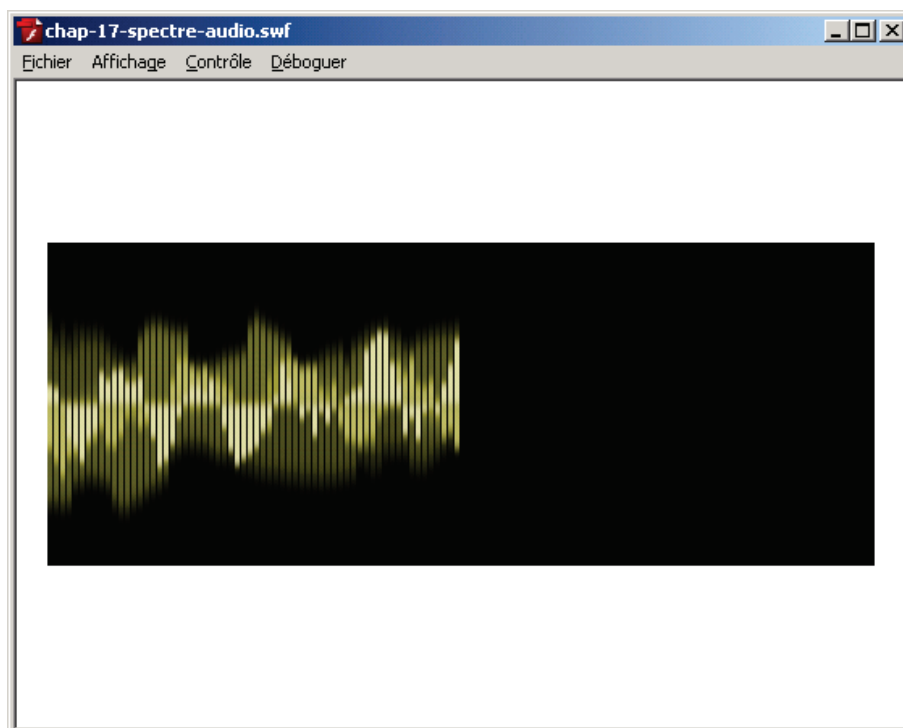
// création d'un égaliseur de 512 pixels de largeur et 200 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 200, 0xDDDDA5 );

addChild ( monEqualiseur );

// centrage de l'égaliseur
monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
```

```
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

La figure 17-9 illustre le résultat :



*Figure 17-9. Spectre altéré par une avec modification de la balance horizontale.*

Nous voyons que notre spectre est fidèle au flux renvoyé par la méthode `computeSpectrum`. En réalité, lorsqu'une transformation est appliquée au son, le spectre renvoyé par la méthode `computeSpectrum` est lui aussi modifié.

Dans le code suivant, nous appliquons un effet à l'aide de la classe `AutoBalance` développée auparavant :

```
// import de la classe Equaliseur
import org.bytearray.media.spectres.Equaliseur;
// import de la classe Amplificateur
import org.bytearray.media.Amplificateur;
// import de la classe AutoBalance
import org.bytearray.media.effets.AutoBalance;

// création d'un objet son et chargement d'un mp3
var monSon:Sound = new Sound ( new URLRequest ("son.mp3") );

// démarrage du son
var canalSon:SoundChannel = monSon.play();

// création de l'objet Amplificateur
var monAmplificateur:Amplificateur = new Amplificateur ( canalSon );
```

```
// création d'un effet de balance horizontale pendant 3 secondes à vitesse
réduite
var effetBalance:AutoBalance = new AutoBalance ( 30, .1 );

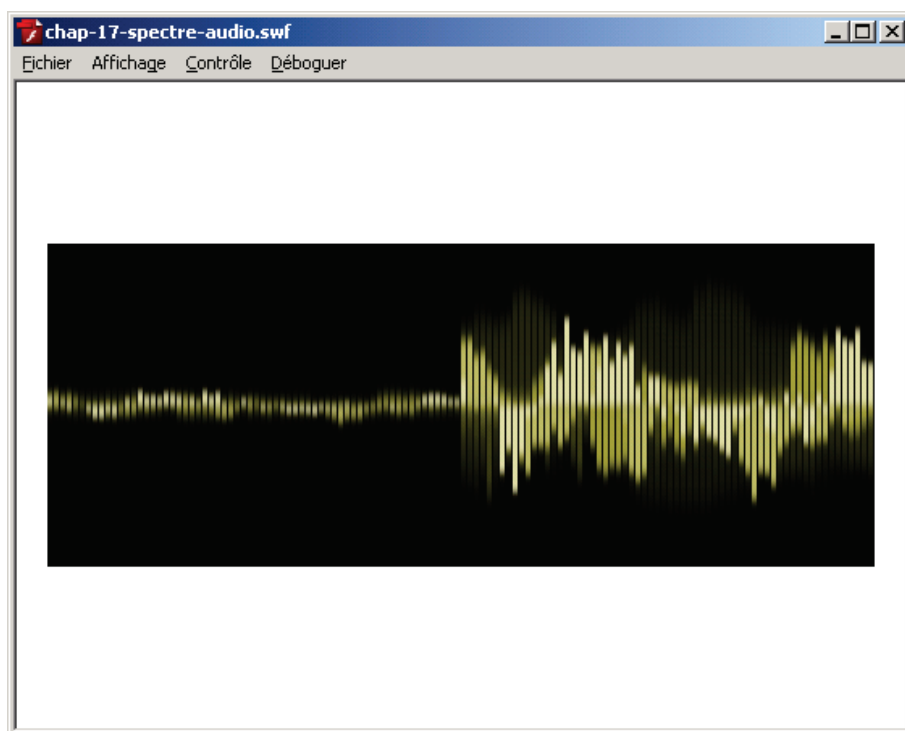
// application de l'effet
monAmplificateur.appliqueEffet ( effetBalance );

// création d'un égaliseur de 512 pixels de largeur et 200 pixels de hauteur
var monEqualiseur:Equaliseur = new Equaliseur( 512, 200, 0xDDDDA5 );

addChild ( monEqualiseur );

// centrage de l'égaliseur
monEqualiseur.x = (stage.stageWidth - monEqualiseur.width) >> 1;
monEqualiseur.y = (stage.stageHeight - monEqualiseur.height) >> 1;
```

En appliquant un effet de balance automatique progressif, le spectre est modifié en temps réel, la figure 17-10 illustre le rendu :



*Figure 17-10. Spectre altéré par une modification progressive de la balance horizontale.*

La méthode `computeSpectrum` nous réserve encore quelques surprises, c'est ce que nous allons découvrir à présent.

## A retenir

- La méthode `computeSpectrum` de la classe `SoundMixer` permet de calculer le spectre de la totalité des sons en cours de lecture.
- La méthode `computeSpectrum` renvoie 512 valeurs.
- La propriété `position` de l'objet `ByteArray` permet de se déplacer manuellement au sein des octets.
- Si une transformation est appliquée à un son, le flux renvoyé par la méthode `computeSpectrum` reflète cette transformation.

## Transformée de Fourier

Comme nous l'avons vu lors du détail des différents paramètres de la méthode `computeSpectrum`, il est possible d'appliquer une transformée de Fourier au spectre.

En activant celle-ci, le spectre généré reflète alors les fréquences des sons en cours de lecture au lieu de l'onde sonore.

Nous allons ajouter deux propriétés constantes au sein de la classe `Equaliseur` afin de pouvoir facilement choisir entre un égaliseur de fréquences ou d'onde sonore.

Pour cela, nous définissons trois propriétés `SPECTRE` et `FREQUENCE` et `fourier` :

```
package org.bytearray.media.spectres

{

    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;
    import flash.filters.BlurFilter;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.geom.ColorTransform;
    import flash.utils.ByteArray;
    import flash.media.SoundMixer;
    import org.bytearray.ouils.BitmapOutils;

    public class Equaliseur extends Bitmap
    {

        public static const SPECTRE:Boolean = false;
        public static const FREQUENCE:Boolean = true;

        private var largeur:int;
        private var hauteur:int;
        private var couleur:Number;
        private var fluxSpectre:ByteArray;
        private var i:int;
        private var amplitude:Number;
        private var surface:Rectangle;
```

```
private var decalage:int;
private var filtreFlou:BlurFilter;
private var point:Point;
private var transformationCouleur:ColorTransform;
private var fourier:Boolean;

public function Equaliseur ( pLargeur:Number, pHauteur:Number,
pCouleurSpectre:Number, pFourier:Boolean=false )
{
    largeur = pLargeur;
    hauteur = pHauteur;
    couleur = pCouleurSpectre;
fourier = pFourier;

    surface = new Rectangle ( 0, 0, 3, 4 );

    fluxSpectre = new ByteArray();

    filtreFlou = new BlurFilter ( 0, 4, 4 );

    point = new Point();

    var composants:Object = BitmapUtils.hexRgb ( pCouleurSpectre );

    transformationCouleur = new ColorTransform (
composants.rouge/255, composants.vert/255, composants.bleu/255 );

    addEventListener ( Event.ADDED_TO_STAGE, activation );
    addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
}

private function activation ( pEvt:Event ):void
{
    bitmapData = new BitmapData ( largeur, hauteur, false, 0 );

    addEventListener ( Event.ENTER_FRAME, calculSpectre );
}

private function desactivation ( pEvt:Event ):void
{
    bitmapData.dispose();
}

private function calculSpectre ( pEvt:Event ):void
{
    SoundMixer.computeSpectrum( fluxSpectre, fourier );

    i = bitmapData.width / 4;

    decalage = 2048 / i;
```



```
while ( i-- )
{
    fluxSpectre.position = i * decalage;

    amplitude = fluxSpectre.readFloat() * ((hauteur - 10) >> 1);

    surface.x = i * 4;

    if ( amplitude > 0 )
    {
        surface.y = (bitmapData.height >> 1) - amplitude;
        surface.height = amplitude;

    } else
    {
        surface.y = (bitmapData.height >> 1);
        surface.height = -amplitude;

    }

    bitmapData.fillRect ( surface, 0xFFFFFFFF );

}

bitmapData.applyFilter ( bitmapData, bitmapData.rect, point,
filtreFlou );

bitmapData.colorTransform ( bitmapData.rect,
transformationCouleur );

}

}
```

Il est important de noter que dans le cas de l'utilisation de la transformée de Fourier, les valeurs renvoyées par la méthode `computeSpectrum` oscillent entre 0 et 1. Nous obtiendrons dans ce cas des bâtonnets dans la partie supérieure du spectre uniquement.

Grâce aux deux propriétés nous pouvons facilement spécifier le type d'équaliseur voulu. Dans le code suivant nous créons un spectre permettant d'afficher les fréquences des sons :

```
// création d'un équaliseur avec transformation de fourier
var monEqualiseur:Equaliseur = new Equaliseur( 512, 200, 0xDDDDA5, Equaliseur.
FREQUENCE );
```

La figure 17-11 illustre le résultat :

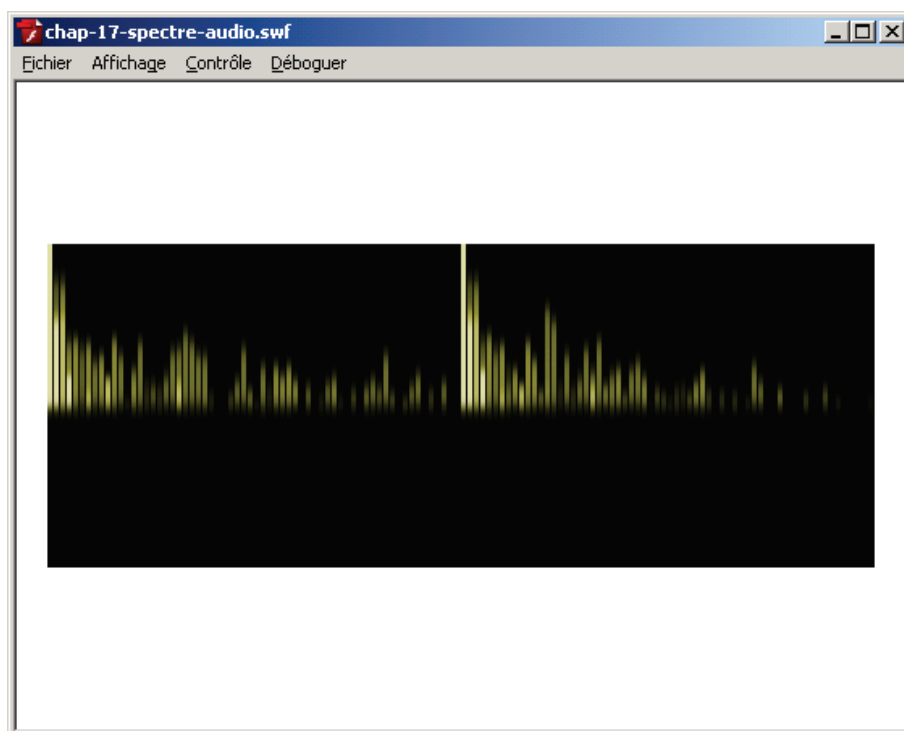


Figure 17-11. Equaliseur avec transformation de fourier.

## A retenir

- La transformée de Fourier permet d'isoler les fréquences des sons en cours de lecture.

## Le format MPEG-4 Audio

La version 9.0.115 du lecteur Flash 9 intègre une compatibilité MPEG-4 et permet la lecture de fichiers audio encodés avec l'algorithme *Advanced Audio Coding* plus communément appelé AAC.

Développé à l'origine par l'institut Fraunhofer, ce format compressé est considéré comme le remplaçant du célèbre codec de compression MP3. A qualité d'écoute égale, le format AAC est environ 30% plus léger que le format MP3.

Cette optimisation du poids des fichiers audio permet donc une réduction de la bande passante utilisée par les sons sur un site à large trafic. Des portails audio comme *iTunes* utilisent déjà le format AAC comme format de distribution. Des périphériques tels le *i-Phone*, la *PlayStation Portable* ainsi qu'un grand nombre de téléphones portables sont aussi compatibles avec ce format.

Le tableau suivant recense les différentes extensions de fichiers MPEG-4 permettant de contenir du son au format AAC compatible avec le lecteur Flash 9 :

Extension	Description
.M4A	Fichier audio
.M4V	Fichier video i-Tunes
.AAC	Fichier audio AAC
.3GP	Fichier audio et vidéo utilisé sur les téléphones 3G
.MP4	Fichier video

*Tableau 1. Extensions de fichiers conteneur du format AAC.*

Notons que le lecteur Flash ne gère pas la lecture de fichiers AAC contenant une piste MP3, ni les fichiers AAC protégés téléchargés depuis des plates-formes telles I-Tunes. De la même manière, les fichiers audio AAC protégés par la technologie de gestion des droits numériques *FairPlay* ne sont pas compatibles.

Dans le code suivant nous chargeons un fichier son MPEG-4 AAC :

```
// instantiation d'un objet NetConnection
var chargeurSon:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurSon.connect(null);

// création d'un objet NetStream
var fluxAudio:NetStream = new NetStream ( chargeurSon );

// lecture du son
fluxAudio.play ("son.m4a");
```

Bien que cela puisse vous surprendre, sachez que la lecture de fichiers audio au format AAC n'est pas assurée par la classe `Sound` mais par les classes `NetStream` et `NetConnection`.

Celles-ci sont utilisées dans le cas d'applications connectée à un serveur de type Flash Media Server ou dans le cas de lecture de vidéos au format FLV.

En testant le code précédent, le fichier son est lu mais l'erreur suivante est levée et affichée dans la fenêtre de sortie :

```
Error #2044: AsyncErrorEvent non pris en charge : text=Error #2095:
flash.net.NetStream n'a pas été en mesure d'appeler l'élément de rappel
onMetaData.
```

Dans le cas de chargement de fichiers à l'aide de la classe `NetStream`, il convient de toujours définir la propriété `client` avant d'appeler la méthode `play`.

La propriété `client`, permet de préciser l'objet sur lequel est définie la méthode `onMetaData`. Aussi étrange que cela puisse paraître, l'objet `NetStream` ne diffuse pas d'événement lié aux métadonnées du média chargé. Nous retrouvons ci le modèle événementiel présent en ActionScript 1 et 2.

Dans le code suivant, nous utilisons le scénario principal comme client :

```
// instantiation d'un objet NetConnection
var chargeurSon:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurSon.connect(null);

// création d'un objet NetStream
var fluxAudio:NetStream = new NetStream ( chargeurSon );

// lecture du son
fluxAudio.play ("son.m4a");

// le scénario joue le rôle du client
fluxAudio.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    /*
    duration : 395.90022675736964
    trackinfo : [object Object]
    audiochannels : 2
    aacaot : 2
    audiosamplerate : 44100
    tags :
    moovposition : 40
    audiocodecid : mp4a
    */
    for ( var p in pMeta ) trace( p + " : " + pMeta[p] );

}
```

Le paramètre `pMeta` reçoit un objet contenant différentes propriétés liées aux métadonnées du média chargé.

Voici en détail chacune des propriétés :

- `aacaot` : le type de fichier audio AAC, cette propriété peut avoir la valeur 0 pour AAC Main, 1 pour AAC LC et 2 pour SBR audio types.
- `audiochannels` : le nombre de canaux du média chargé. Dans le cas de fichiers audio AAC multicanaux, ces derniers sont décodés sur deux canaux seulement par le lecteur Flash.
- `audiocodecid` : le codec audio utilisé du média chargé. La chaîne de caractères `mp4a` est utilisée pour le format AAC, et `.mp3` pour les fichiers MP3.

- `audiosamplerate` : fréquence d'échantillonnage du fichier audio.
- `duration` : la durée en secondes du média chargé.
- `moovposition` : La position de l'atome moov au sein du média chargé.
- `tags` : un objet comprenant différentes informations liées au média chargé. L'équivalent des données ID3 du format MP3.
- `trackinfo` : un objet contenant les eventuelles illustrations liées au média (pochettes, photos) sous la forme de `ByteArray`.

Au cas où le fichier MPEG-4 n'est pas compatible nous pouvons écouter l'événement `NetStatusEvent.NET_STATUS` :

```
// instantiation d'un objet NetConnection
var chargeurSon:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurSon.connect(null);

// création d'un objet NetStream
var fluxAudio:NetStream = new NetStream ( chargeurSon );

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxAudio.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

function etatLecture ( pEvt:NetStatusEvent ):void

{

    if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

    else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
trace("aucune piste trouvée");

}

// lecture du son
fluxAudio.play ("son.m4a");

// le scénario joue le rôle du client
fluxAudio.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    /*
    duration : 395.90022675736964
    trackinfo : [object Object]
    audiochannels : 2
    aacaot : 2
    audiosamplerate : 44100
    tags :
    moovposition : 40
    audiocodecid : mp4a
    */
    for ( var p in pMeta ) trace( p + " : " + pMeta );
```

```
| }  
}
```

L'objet événementiel diffusé par l'événement

`NetStatusEvent.NET_STATUS` possède une propriété `info` contenant un objet disposant d'informations sur l'état de la connexion.

Cet objet possède les deux propriétés suivantes :

- `code` : une chaîne de caractères indiquant l'état de la connexion. Consultez la documentation pour connaître les différentes valeurs renvoyées.
- `level` : renvoie la chaîne de caractère `status` si la connexion est réussie ou `error` si celle-ci échoue.

Malheureusement, il n'existe pas de propriétés constantes de classe afin de tester les valeurs retournées par les propriétés `code` et `level`.

Nous venons de terminer notre aventure sonore, nous allons nous intéresser dans la partie suivante aux différentes nouveautés apportées par ActionScript 3 et la dernière version du lecteur Flash 9 en matière de vidéo.

## A retenir

- La version 9.0.115 du lecteur Flash 9 intègre un décodage des fichiers audio AAC.
- L'algorithme de compression AAC est considéré comme plus performant. C'est à beaucoup d'égards un remplaçant supérieur au format MP3.
- Afin de lire un fichier audio AAC, nous utilisons les classes `NetConnection` et `NetStream`.
- La propriété `client` de l'objet `NetStream` permet de définir l'objet interceptant l'événement `onMetaData`.
- L'événement `NetStatusEvent.NET_STATUS` diffusé par l'objet `NetStream` permet de savoir si une erreur de décodage est intervenue.
- La lecture de fichiers MPEG-4 fonctionne en ActionScript 1, 2 et 3.

## La vidéo dans Flash

Le lecteur Flash s'est imposé aujourd'hui comme lecteur multimédia incontournable sur Internet. En plus d'offrir un décodage audio MPEG-4, la dernière version du lecteur Flash révolutionne la vidéo sur réseaux en intégrant une compatibilité avec le codec de compression vidéo MPEG-4 H.264.

Depuis sa version 9.0.115, le lecteur Flash 9 intègre donc les 4 codecs suivants :

- Sorenson Spark : il s'agit du premier codec vidéo à être apparu dans le lecteur Flash. La qualité d'affichage n'est pas optimale, ce codec est voué à disparaître.
- Screen Video : Il s'agit du codec utilisé pour la capture d'écran par Connect (anciennement Breeze).
- On2 VP6 : ce codec fut introduit au sein du lecteur Flash 8. Il introduit une qualité d'image supérieure ainsi que la gestion du canal alpha.
- H.264 : ce codec fut introduit au sein du lecteur Flash 9.0.115. Il améliore à nouveau la qualité de l'image tout en garantissant l'interopérabilité et l'universalité des données vidéo.

Voici la liste des différentes classes impliquées dans la lecture de flux vidéo au sein du lecteur Flash :

- `flash.net.NetConnection` : La classe `NetConnection` permet d'ouvrir la connexion.
- `flash.net.NetStream` : La classe `NetStream` permet de manipuler le flux en cours de lecture.
- `flash.media.Video` : La classe `Video` permet d'afficher le flux chargé.

Passons à la pratique, dans cette nouvelle partie nous allons découvrir comment charger et lire une vidéo MPEG-4 de manière dynamique.

## Le format MPEG-4 Video

Pour lire une vidéo MPEG-4 nous utilisons les classes `NetStream` et `NetConnection`, de la même manière qu'une vidéo au format FLV. Sachez que le lecteur Flash ne s'appuie pas sur les extensions de fichiers audio ou video afin de tester le type de la vidéo mais sur l'entête (binaire) du fichier en question.

Dans le cas d'une ancienne application censée charger des fichiers vidéo au format FLV, il est tout à fait possible de renommer l'extension d'une vidéo MPEG-4 comme `mov` ou `mp4` en `flv`, celle-ci sera lue sans problèmes.

Dans le code suivant nous chargeons une vidéo MPEG-4 stockée dans un fichier QuickTime `mov` :

```
// instantiation d'un objet NetConnection
var chargeurVideo:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurVideo.connect(null);

// création d'un objet NetStream
```

```
var fluxVideo:NetStream = new NetStream ( chargeurVideo );

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxVideo.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

// lecture du fichier vidéo MPEG-4
fluxVideo.play ( "video_hd.mov" );

// le scénario joue le rôle du client
fluxVideo.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{

    /*
    trackinfo : [object Object],[object Object],[object Object]
    audiochannels : 2
    width : 640
    videoframerate : 23.976
    height : 268
    duration : 95.15537414965986
    videocodecid : avc1
    audiosamplerate : 44100
    seekpoints : [object Object],[object Object],[object Object]
    moovposition : 40
    avcprofile : 77
    aacaot : 2
    audiocodecid : mp4a
    avclevel : 21
    */
    for ( var p in pMeta ) trace( p + " : " + pMeta[p] );

}

function etatLecture ( pEvt:NetStatusEvent ):void

{

    if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

    else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
trace("aucune piste trouvée");

}
```

L'objet passé à la méthode `onMetaData` possède des propriétés quelque peu différentes de la lecture d'un fichier audio AAC.

Voici le détail de chacune des propriétés :

- `aacaot` : le type de fichier audio AAC, cette propriété peut avoir la valeur 0 pour AAC Main, 1 pour AAC LC et 2 pour SBR audio types.
- `audiochannels` : le nombre de canaux du média chargé. Dans le cas de fichiers audio AAC multicanaux, ces derniers sont décodés sur deux canaux seulement par le lecteur Flash.



- `audiocodecid` : le codec audio utilisé du média chargé. La chaîne de caractères `mp4a` est utilisée pour le format AAC, et `.mp3` pour les fichiers MP3.
- `audiosamplerate` : fréquence d'échantillonnage du fichier audio.
- `avclevel` : cette propriété renvoie un nombre compris entre 10 et 51 donnant des informations au décodeur concernant les ressources nécessaires pour le décodage de la vidéo.
- `avcprofile` : le profil du fichier H.264, une valeur pouvant être 66, 77, 88, 100, 110, 122 ou 144.
- `duration` : la durée en secondes du média chargé.
- `height` : la hauteur en pixels de la vidéo.
- `moovposition` : la position de l'atome moov au sein du média chargé.
- `seekpoints` : points de repères permettant le chapitrage du média.
- `tags` : un objet comprenant différentes informations liées au média chargé. L'équivalent des données ID3 du format MP3.
- `trackinfo` : un objet contenant différentes informations liées au média chargé.
- `videoframerate` : un objet contenant différentes informations liées au média chargé.
- `videocodecid` : un objet contenant différentes informations liées au média chargé.
- `width` : la largeur en pixels de la vidéo.

Nous venons de voir comment charger une vidéo dynamiquement, il nous faut maintenant l'afficher. Pour cela, nous devons lier le flux de l'objet `NetStream` à un objet `Video`.

### A retenir

- Les fichiers vidéo MPEG-4 sont lus à l'aide des classes `NetConnection` et `NetStream`.
- L'introduction du codec MPEG-4 n'empêche pas la lecture de vidéos au format FLV.

### La classe Video

La classe `Video` réside au sein du paquetage `flash.media`. Contrairement aux précédentes versions d'ActionScript, celle-ci est désormais instanciable par programmation.

La classe `Video` hérite de la classe `DisplayObject`, et possède donc toutes les propriétés et méthodes propres à un objet graphique.

Grâce à la méthode `attachNetStream` nous pouvons lier le flux vidéo à l'objet `Video` :

```
// instantiation d'un objet NetConnection
var chargeurVideo:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurVideo.connect(null);

// création d'un objet NetStream
var fluxVideo:NetStream = new NetStream ( chargeurVideo );

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxVideo.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

// création de l'objet Video
var ecranVideo:Video = new Video();

// on attache le flux à l'écran vidéo
ecranVideo.attachNetStream( fluxVideo );

// ajout à la liste d'affichage
addChild ( ecranVideo );

// lecture du fichier vidéo MPEG-4
fluxVideo.play ( "wall-e-tsrl_h.640.mov" );

// le scénario joue le rôle du client
fluxVideo.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void
{
    for ( var p in pMeta ) trace( p + " : " + pMeta[p] );
}

function etatLecture ( pEvt:NetStatusEvent ):void
{
    if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

    else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
trace("aucune piste trouvée");
}
```

Le code précédent génère le résultat illustré par la figure 17-12 :

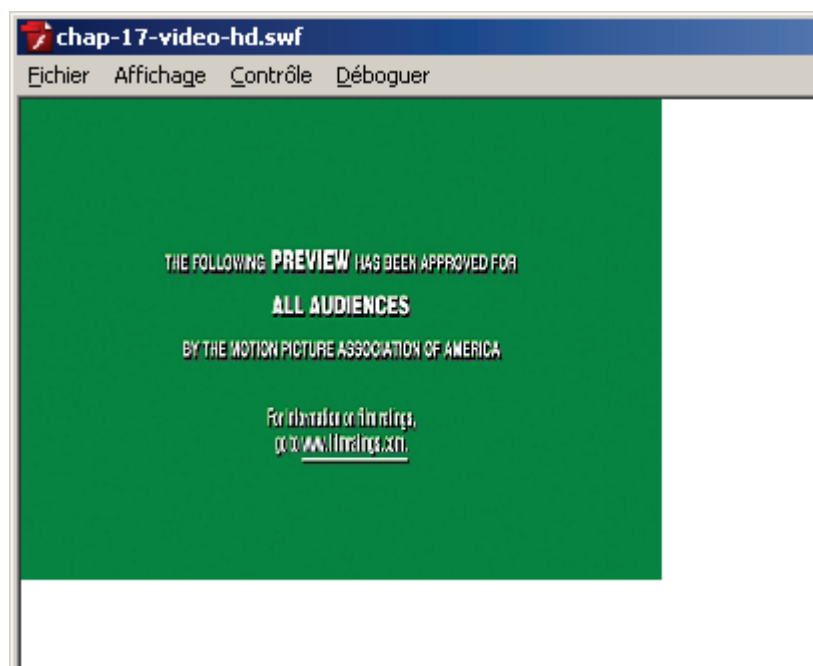


Figure 17-12. Lecture de vidéo MPEG-4.

L'objet `Video` ne se redimensionne pas automatiquement aux dimensions du média. Celui-ci possède une largeur de 320 pixels en largeur et 240 pixels en hauteur.

Nous allons utiliser les propriétés `width` et `height` de l'objet passé à la méthode `onMetaData` afin de redimensionner l'objet `Video` :

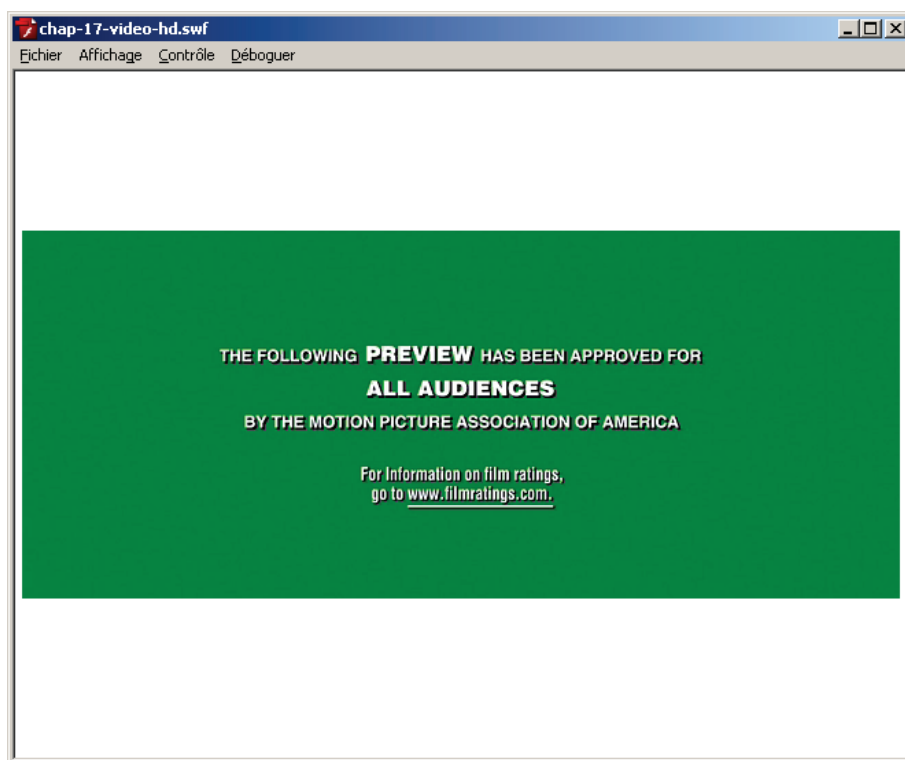
```
// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void
{
    ecranVideo.width = pMeta.width;
    ecranVideo.height = pMeta.height;

    if ( !contains ( ecranVideo ) )
    {
        addChild ( ecranVideo );
    }

    ecranVideo.x = (stage.stageWidth - ecranVideo.width) >> 1;
    ecranVideo.y = (stage.stageHeight - ecranVideo.height) >> 1;
}
```

Nous ajoutons l'objet `Video` à la liste d'affichage au sein de la méthode `onMetaData`. Nous testons si l'objet `Video` n'est pas déjà présent à l'affichage, le cas échéant nous l'ajoutons, puis nous centrons la vidéo.

La figure 17-13 illustre le résultat :



*Figure 17-13. Vidéo MPEG-4 adaptée et centrée.*

Afin de conserver une image lissée, nous pouvons activer la propriété `smoothing` de l'objet `Video` dont la valeur par défaut est à `false`.

Il est conseillé d'activer cette propriété lors de la lecture de vidéos au sein du lecteur Flash 9.0.115 et versions ultérieures afin de tirer profit de l'optimisation de l'image par *mip-mapping*.

---

Attention toutefois l'utilisation conjointe du mode plein écran et de cette propriété, peut engendrer une surcharge du processeur importante sur les machines peu puissantes.

---

Pour plus d'informations concernant le *mip-mapping*, consultez le chapitre 12 intitulé *Programmation Bitmap*.

## A retenir

- En ActionScript 3 la classe `Video` est instanciable par programmation.
- L'objet `Video` est lié au flux chargé à l'aide de la méthode `attachNetStream`.

## Transformation du son lié à un objet `NetStream`

Nous avons en début de chapitre comment modifier le son à l'aide de la propriété `soundTransform` de l'objet `SoundChannel`. Lorsqu'un média est lu à l'aide de la classe `NetStream`, aucun objet `SoundChannel` n'est disponible.

L'objet `SoundTransform` lié au média en cours de lecture est accessible par la propriété `soundTransform` de l'objet `NetStream`.

Dans le code suivant, nous réduisons le volume de la vidéo de 50 % :

```
// instantiation d'un objet NetConnection
var chargeurVideo:NetConnection = new NetConnection();

// lors d'un chargement de fichier local nous nous connectons à null
chargeurVideo.connect(null);

// création d'un objet NetStream
var fluxVideo:NetStream = new NetStream ( chargeurVideo );

// récupération de l'objet SoundTransform associé au média chargé
var transformation:SoundTransform = fluxVideo.soundTransform;

// modification du volume
transformation.volume = .5;

// application de la modification
fluxVideo.soundTransform = transformation;

// écoute de l'événement NetStatusEvent.NET_STATUS
fluxVideo.addEventListener( NetStatusEvent.NET_STATUS, etatLecture );

// création de l'objet Video
var ecranVideo:Video = new Video();

// on attache le flux à l'écran vidéo
ecranVideo.attachNetStream( fluxVideo );

// ajout à la liste d'affichage
addChild ( ecranVideo );

// lecture du fichier vidéo MPEG-4
fluxVideo.play ( "wall-e-tsrl_h.640.mov" );

// le scénario joue le rôle du client
fluxVideo.client = this;

/// méthode onMetaData définie sur le scénario principal
function onMetaData ( pMeta ):void

{
```

```
        for ( var p in pMeta ) trace( p + " : " + pMeta[p] );
    }

    function etatLecture ( pEvt:NetStatusEvent ):void
    {
        if ( pEvt.info.code == "NetStream.FileStructureInvalid" ) trace("fichier
non compatible");

        else if ( pEvt.info.code == "NetStream.NoSupportedTrackFound" )
            trace("aucune piste trouvée");
    }
}
```

Le même code s'applique dans le cas du chargement d'un son MPEG-4 AAC.

## A retenir

- Afin de modifier le son d'un média associé à un objet `NetStream` nous utilisons sa propriété `soundTransform`.

## Mode plein-écran

Afin de bénéficier pleinement du décodage video MPEG-4 du lecteur Flash 9, celui-ci intègre en plus depuis la version 9.0.28 la capacité de passer l'affichage en mode plein écran.

Jusqu'à présent, cette fonctionnalité n'était réservée qu'au lecteur Flash autonome qui permettait le passage en mode plein écran par le biais du mode projecteur.

Dans le cas de moniteurs multiples, l'écran ayant le plus de contenu Flash en cours d'affichage est choisi automatiquement par le lecteur.

La classe `Stage` définit une propriété `displayState` permettant le passage du lecteur en plein écran, celle-ci accepte deux valeurs stockées au sein de la classe `StageDisplayState` :

- `StageDisplayState.FULL_SCREEN` : Mode plein écran.
- `StageDisplayState.NORMAL` : Mode normal.

Attention, le passage en mode plein écran est soumis aux différentes restrictions suivantes :

- Le mode plein-écran ne peut pas être déclenché de manière autonome. Seule une action utilisateur clavier ou souris permet d'activer le mode plein écran. Dans le cas contraire une erreur est levée à l'exécution.
- La page contenant le lecteur Flash doit autoriser le mode plein-écran en activant l'attribut `allowFullScreen` des balises `<embed>` et

`<object>` à l'aide du booléen `true`. La valeur par défaut est `false` ce qui empêche l'activation du mode plein-écran.

- Les touches du clavier sont verrouillées à l'exception de la touche ESC permettant de repasser en mode normal, la saisie de texte est donc impossible.

Dans le code suivant, nous passons en mode plein-écran lorsque l'utilisateur clique sur la scène :

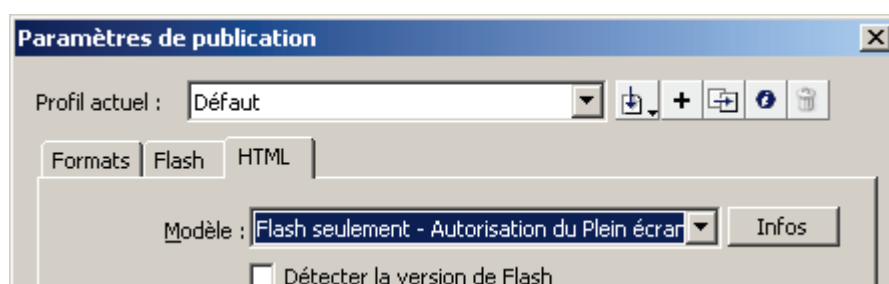
```
// écoute de l'événement MouseEvent.CLICK auprès de l'objet Stage
stage.addEventListener ( MouseEvent.CLICK, gestionAffichage );

function gestionAffichage ( pEvt:MouseEvent ):void
{
    // passage en mode plein-écran
    stage.displayState = StageDisplayState.FULL_SCREEN;
}
```

Par défaut le code précédent n'est pas suffisant, afin d'autoriser le mode plein écran l'animation doit être lue au sein du lecteur Flash du navigateur et l'attribut `allowFullScreen` des balises `<embed>` et `<object>` doit être passé à `true`.

Afin d'automatiser l'activation du mode plein écran au sein de la page conteneur, il est conseillé de sélectionner au sein de l'onglet HTML de panneau *Paramètres de publication* le modèle *Flash seulement autorisation du plein écran*.

La figure 17-14 illustre le modèle prédéfini :



*Figure 17-14 : Onglet HTML du panneau Paramètres de publication.*

Une nouvelle propriété `fullScreenSourceRect` fut introduite au sein du lecteur Flash 9.0.115. Celle-ci permet de spécifier la surface à passer en plein écran. Ce paramètre est idéal pour déterminer quelle partie de l'application doit être redimensionnée.

Pour l'utiliser nous devons créer une instance de la classe `flash.geom.Rectangle` afin de définir la surface voulue :

```
// écoute de l'événement MouseEvent.CLICK auprès de l'objet Stage
```

```
stage.addEventListener ( MouseEvent.CLICK, gestionAffichage );

function gestionAffichage ( pEvt:MouseEvent ):void
{
    // une surface est définie comme zone à passer en plein écran
    var surfacePleinEcran:Rectangle = new Rectangle ( 0, 0, 150, 150 );

    // la zone est spécifiée
    stage.fullScreenSourceRect = surfacePleinEcran;

    // passage en mode plein-écran
    stage.displayState = StageDisplayState.FULL_SCREEN;
}
```

Ainsi, nous pouvons choisir comme zone à agrandir la surface occupée par la vidéo en cours de lecture :

```
// écoute de l'événement MouseEvent.CLICK auprès de l'objet Stage
stage.addEventListener ( MouseEvent.CLICK, goFull );

function goFull ( pEvt:MouseEvent ):void
{
    // la surface occupée par la vidéo est définie comme zone à agrandir
    var surfacePleinEcran:Rectangle = new Rectangle ( ecranVideo.x,
    ecranVideo.y, ecranVideo.width, ecranVideo.height );

    // la zone est spécifiée
    stage.fullScreenSourceRect = surfacePleinEcran;

    // passage en mode plein-écran
    stage.displayState = StageDisplayState.FULL_SCREEN;
}
```

Afin de faciliter ce processus de redimensionnement, le lecteur Flash peut allouer cette tâche au processeur de la carte graphique afin de rendre le redimensionnement moins gourmand et plus fluide.

Le lecteur Flash s'appuie sur Direct X sous Windows et Open GL sur Mac OS X. Au cas où la carte graphique ne serait pas compatible, une accélération dite logicielle est appliquée, le processeur est donc chargée de la tâche.

Afin d'activer l'accélération matérielle, il suffit de sélectionner l'option *Paramètres* de la liste déroulante du lecteur Flash et de cocher la case *Activer l'accélération matérielle* au sein de l'onglet *Affichage* illustré par la figure 17-15 :





*Figure 17-15 : Onglet Affichage du lecteur Flash.*

Il n'est pas possible d'activer ou désactiver l'accélération matérielle par programmation.

## A retenir

- La classe `Stage` définit une propriété `displayState` pouvant prendre comme valeur `StageDisplayState.NORMAL` et `StageDisplayState.FULL_SCREEN`.
- Le mode plein écran ne peut être déclenché de manière autonome.
- Le mode plein écran doit être autorisé au sein de la page conteneur grâce à l'attribut `allowFullScreen`.
- Les touches du clavier sont verrouillées, à l'exception de la touche ESC. La saisie du texte est impossible, ce qui limite malheureusement l'exploitation du mode plein-écran.

ActionScript 3 nous réserve encore des surprises, au cours du prochain chapitre nous allons découvrir un nouveau moyen de communiquer avec l'extérieur grâce aux connexions par socket.

# 18

## Sockets

<b>UNE PASSERELLE UNIVERSELLE .....</b>	<b>1</b>
<b>CREER UN SERVEUR DE SOCKET XML.....</b>	<b>6</b>
LA CLASSE XMLSocket .....	7
CREER UN T’CHAT MULTI-UTILISATEUR .....	11
<b>LA CLASSE SOCKET .....</b>	<b>23</b>
CREER UN SERVEUR DE SOCKET BINAIRE .....	24
ECHANGER DES DONNÉES .....	28

### Une passerelle universelle

Nous avons vu au cours des chapitres précédents comment le lecteur Flash pouvait communiquer avec l’extérieur. Nous allons aller plus loin en abordant à présent la notion de communication par socket.

Une socket doit être considérée comme une passerelle de communication entre deux applications fonctionnant sur un réseau. Dans la vie courante, une socket pourrait être assimilée à une connexion téléphonique entre deux personnes.

Pour entreprendre une communication par socket, deux acteurs au minimum sont nécessaires :

- Le serveur : le serveur de socket écoute les connexions entrantes, il se charge de gérer les clients connectés et communique avec eux.
- Le client : le client se connecte au serveur par la socket et communique avec le serveur.

Notons que l’avantage principal d’une connexion par socket réside dans le caractère persistant de la communication entre les deux acteurs. Contrairement à une connexion HTTP traditionnelle, la socket

maintient la connexion ouverte entre les deux acteurs et permet au serveur de transmettre des informations aux clients sans que ces derniers n'en fassent la demande.

Ce comportement offre ainsi de nombreuses possibilités en termes d'applications dynamiques temps réel.

Afin de bien comprendre la notion de socket, prenons l'exemple suivant :

Lorsque vous naviguez sur Internet à l'aide votre navigateur favori, ce dernier demande au serveur distant de lui transmettre la page que vous souhaitez afficher. Ce langage commun entre les deux acteurs est appelé *protocole d'application* et fait partie de ce que l'on appelle la *suite de protocoles Internet*.

---

Internet est basé sur cet ensemble de protocole  
généralement appelé modèle TCP/IP.

---

Afin d'afficher une page spécifique, le navigateur envoie au serveur HTTP la requête suivante :

```
GET /index.php HTTP/1.1  
Host: www.bytearray.org
```

Nous pouvons remarquer la présence de mots réservés tels `GET` ou `Host` faisant partie du protocole d'application HTTP. Par ces mots clés, le serveur comprend qu'il doit transmettre le contenu de la page `index.php` au client connecté, ici notre navigateur.

Aussitôt la requête reçue, le serveur l'analyse et répond à l'aide du message suivant :

```
HTTP/1.x 200 OK  
Date: Sun, 20 Jan 2008 14:30:34 GMT  
Server: Apache/2.0.54 (Debian GNU/Linux) PHP/4.3.10-22 mod_ssl/2.0.54  
OpenSSL/0.9.7e mod_perl/1.999.21 Perl/v5.8.4  
X-Powered-By: PHP/4.3.10-22  
X-Pingback: http://www.bytearray.org/xmlrpc.php  
Content-Encoding: gzip  
Vary: Accept-Encoding  
Content-Length: 4260  
Keep-Alive: timeout=15, max=97  
Connection: Keep-Alive  
Content-Type: text/html; charset=UTF-8
```

En réalité, l'application serveur et le client communiquent par socket à l'aide d'un protocole commun. Un nombre illimité de protocoles d'applications peuvent ainsi être implémentés ou créés puis utilisés à l'aide d'une connexion par socket.

Nous sommes donc en mesure d'utiliser au sein de Flash n'importe quel protocole d'application tel HTTP, FTP, SMTP ou autres.

Pour différencier les applications d'un ordinateur associées à chaque protocole, chaque application est associée à un port spécifique appelé couramment *port logiciel*. Différentes applications sont donc accessibles à partir d'une même adresse IP mais un port unique.

Parmi les protocoles d'applications les plus connus, nous pouvons établir le tableau suivant :

Port	Application
21	FTP (transfert de fichiers)
25	SMTP (envoi de courrier électronique)
80	HTTP (transfert hypertexte)
110	POP (stockage de courrier électronique)

*Tableau 1. Liste de protocoles communs.*

Afin de connaître les ports logiciels associés à chaque protocole. Vous pouvez consulter la liste disponible aux adresses suivantes :

- [http://fr.wikipedia.org/wiki/Liste\\_des\\_ports\\_logiciels](http://fr.wikipedia.org/wiki/Liste_des_ports_logiciels)
- <http://www.iana.org/assignments/port-numbers>

Ainsi, nous pouvons placer les différents ports dans trois catégories :

- Les ports bien connus (0 à 1023) : Ces ports sont couramment utilisés par des processus systèmes ayant un accès privilégié. Il est donc déconseillé d'y avoir recours pour un serveur de socket personnalisé, un risque de collision serait encouru.
- Les ports enregistrés (1024 à 49151) : Ils sont dédiés aux développements d'applications courantes. La plupart de ces ports demeurent libres et peuvent être utilisés.
- Les ports dynamiques privés (49152 à 65535) : Ces ports demeurent généralement libres.

Il est important de préciser que les classes de socket ActionScript 3 s'appuient sur un modèle de transmission TCP et sont considérées comme des passerelles d'échanges sûres, s'opposant au modèle de transmission UDP considéré comme peu fiable.

Mais qu'entendons-nous par passerelle d'échange sûre ?

En réalité, deux types de protocoles peuvent être utilisés sur Internet, le premier de type TCP puis le second de type UDP :

- TCP (Transmission Control Protocol) : Dans un contexte de connexion TCP, les données échangées sont contrôlées. Au cas où la transmission est trop rapide ou si des données sont perdues, le serveur ralentit le débit, et réexpédie les données non correctement transmises.
- UDP (User Datagram Packet) : Dans un contexte de connexion UDP, afin de gagner en vitesse de transfert aucun contrôle n'est effectué lors

du transfert des données. En revanche certains paquets peuvent ne jamais parvenir. Ce type de socket est généralement utilisé dans le cas de jeux en temps réel où la perte de quelques paquets n'influence pas le bon fonctionnement de l'application.

Les classes de socket ActionScript 3 s'appuient sur le protocole TCP, une compatibilité future avec le protocole UDP pourrait être intéressante pour certains types d'applications.

Une communication par socket est généralement divisée en plusieurs phases distinctes :

- 1. Le client se connecte au serveur de socket en envoyant une commande spécifique afin d'initier la conversation ou de s'authentifier.
- 2. Le serveur décide d'accepter ou non le client. Cette étape peut être vérifiée de par la provenance du client ou les informations soumises par ce dernier lors de la procédure d'authentification.
- 3. La communication est établie.

A tout moment, un des acteurs de la communication peut décider de clore la connexion. Le serveur peut décider d'interrompre la communication au cas où un client ne serait plus autorisé ou simplement lorsqu'un client souhaite se déconnecter.

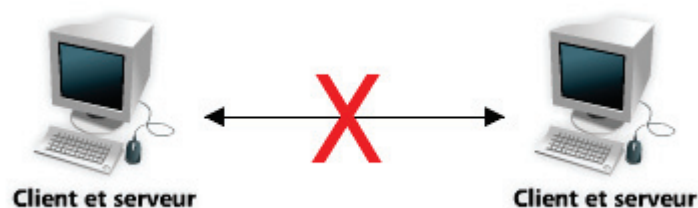
Voici les deux classes permettant de se connecter à un serveur de `Socket` en ActionScript 3 :

- `flash.net.XMLSocket` : la classe `XMLSocket` permet l'échange de données au format XML ou texte.
- `flash.net.Socket` : la classe `Socket` permet l'échange de données brutes au format binaire.

Il est important de noter qu'ActionScript 3 n'intègre pas de classes permettant la création de serveurs de sockets, seules des applications clients pourront être développées.

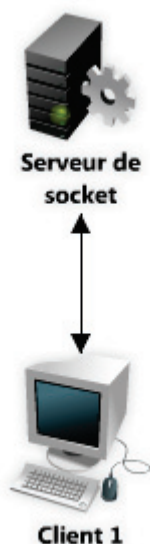
Il est donc impossible de connecter directement deux applications Flash au travers d'une connexion socket. Pour cela, l'une des applications devrait se comporter comme hôte ce qui est impossible en ActionScript 3.

Si tel était le cas, la création d'un réseau *peer 2 peer* entre différentes applications Flash serait envisageable.



*Figure 18-1. Connexion peer 2 peer non supportée.*

Le choix du langage utilisé pour la création du serveur dépend de vos besoins. Des langages tels le C, C++, C#, Java ou encore PHP offrent tous la possibilité de créer des serveurs de socket.



*Figure 18-2. Connexion à un serveur de socket.*

Nous allons nous intéresser dans la partie suivante aux différents types de sockets disponibles en ActionScript 3.

## A retenir

- Une socket est une passerelle de communication entre deux applications évoluant sur un réseau.
- Les classes de Socket ActionScript 3 sont basées sur le modèle TCP/IP.
- Le protocole UDP n'est pas pris en charge.
- Les classes de socket ActionScript 3 ne permettent pas la création de serveur de socket.
- Le nombre de messages échangés à travers une connexion socket est illimité.
- ActionScript 3 intègre deux classes liées aux sockets : `XMLSocket` et `Socket`.

## Créer un serveur de socket XML

Avant d'entamer le développement du serveur de socket, il convient de définir son rôle ainsi que son fonctionnement.

Le rôle d'un serveur de socket est d'écouter les connexions entrantes puis de gérer par la suite la connexion ainsi que les échanges entre les acteurs impliqués dans la communication.

Nous avons vu précédemment qu'il était impossible de connecter directement plusieurs applications Flash entre elles. A l'inverse, le serveur de socket peut agir comme relais afin de faire transiter les messages entre les différents clients.

Vous pensiez peut être que cela était réservé à des technologies telles *Flash Media Server* ou *Red 5*, nous allons voir que quelques lignes de PHP vont nous permettre de connecter plusieurs applications Flash entre elles. Nous allons développer au cours de cette partie un serveur de socket XML afin de créer un t'chat multi utilisateurs.

Comme nous l'avons vu précédemment, de nombreux langages permettent la création de serveur de socket de manière simplifiée. Parmi ceux là nous pouvons citer Java, C#, C++ ou PHP qui s'avère être le moyen le langage le plus simple pour déployer votre serveur de socket.

Une explication approfondie de l'API de socket PHP serait hors sujet, c'est la raison pour laquelle nous ne rentrerons pas dans les détails du code du serveur. Nous allons donc commencer par un simple serveur renvoyant un message de bienvenue lorsque nous nous connectons.

Le code PHP suivant est placé au sein d'un fichier nommé `ServeurXMLSocket.php` :

```
|#!/usr/local/bin/php -q
```

```
<?php  
  
set_time_limit(0);  
  
$adresse = "localhost";  
$port = "10000";  
  
$connexion = socket_create (AF_INET, SOCK_STREAM, SOL_TCP);  
socket_bind ($connexion, $adresse, $port);  
  
socket_listen ($connexion, 1);  
  
echo "Le serveur de socket est en route !";  
  
$client = socket_accept ($connexion);  
  
socket_close ($client);  
  
socket_close ($connexion);  
  
?>
```

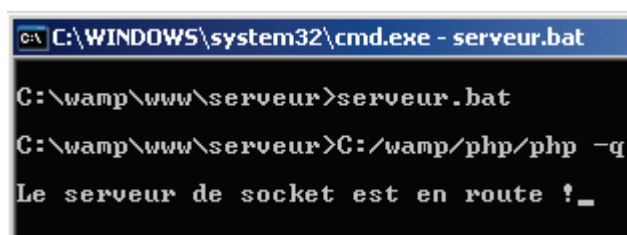
Afin de démarrer convenablement le serveur, il convient de le lancer par ligne de commande.

Pour cela, nous créons un nouveau fichier texte contenant le code suivant :

```
C:/wamp/bin/php/php5.2.5/php.exe -q c:/wamp/www/serveur/ServeurXMLSocket.php
```

Puis, nous sauvons le fichier texte sous le nom `serveur.bat`.

Depuis la ligne de commande nous accédons au fichier `.bat` puis nous l'exécutons, le message illustré par la figure 18-3 doit s'afficher :



*Figure 18-3. Démarrage du serveur depuis la ligne de commande.*

Une fois le serveur démarré, nous devons nous y connecter, c'est ce que nous allons découvrir dans la partie suivante.

## La classe XMLSocket

Afin de pouvoir se connecter à notre serveur de socket, nous pouvons utiliser une instance de la classe `XMLSocket` dont voici le constructeur :



```
| public function XMLSocket(host:String = null, port:int = 0)
```

Voici le détail de chacun des paramètres :

- `host` : il s'agit de l'adresse du serveur de socket. Attention, si le serveur évolue au sein d'un autre domaine que le SWF, la connexion doit être autorisée par un fichier de régulation.
- `port` : le numéro de port TCP utilisé lors de la connexion. Par défaut il est impossible de se connecter à un port inférieur à 1024 pour des raisons de précautions. Si vous souhaitez tout de même utiliser un port inférieur vous devez utiliser un fichier de régulation.

Dans le code suivant, nous établissons une connexion avec notre serveur de socket XML :

```
| // création de la connexion  
| var connexion:XMLSocket = new XMLSocket("localhost", 10000);
```

Au cas où l'adresse et le port utilisé ne seraient pas précisés lors de l'instanciation, nous pouvons utiliser la méthode `connect` :

```
| // création de l'objet XMLSocket  
| var connexion:XMLSocket = new XMLSocket();  
  
| // connexion au serveur de socket  
| connexion.connect("localhost", 10000);
```

Attention, les classes `XMLSocket` et `Socket` n'ont pas la possibilité de se connecter à des ports supérieurs à 65535 et inférieurs à 1024. L'utilisation de ports inférieurs à 1024 pourrait entraîner une collision avec des serveur tels HTTP, FTP ou autres.

---

Si vous souhaitez utiliser un port inférieur à 1024,  
l'utilisation d'un fichier de régulation est nécessaire.

---

Afin d'écouter les différentes phases liées à la connexion nous utilisons les événements diffusés par l'objet `XMLSocket` dont voici la liste :

- `Event.CLOSE` : diffusé lorsque la connexion socket est interrompue.
- `Event.CONNECT` : diffusé lorsque la connexion au serveur a pu être réalisée.
- `DataEvent.DATA` : diffusé lors de l'envoi ou réception de données.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque la connexion au serveur de socket n'a pas aboutie.
- `SecurityError.SECURITY` : diffusé lorsque la connexion socket tente de se connecter auprès d'un serveur non autorisé ou à port inférieur à 1024.

Nous écoutons les différents événements de connexions ainsi que l'événement `DataEvent.DATA` :

```
// création d'une instance de XMLSocket
var connexion:XMLSocket = new XMLSocket("localhost", 10000);

// écoute des événements
connexion.addEventListener ( Event.CONNECT, connexionReussie );
connexion.addEventListener ( Event.CLOSE, fermetureConnexion );
connexion.addEventListener ( DataEvent.DATA, receptionDonnees );

function connexionReussie ( pEvt:Event ):void
{
    trace("connexion réussie");
}

function fermetureConnexion ( pEvt:Event ):void
{
    trace("fermeture de la connexion");
}

function receptionDonnees ( pEvt:DataEvent ):void
{
    trace("réception des données");
}
```

En testant le code précédent, nous voyons que la connexion aboutit, puis nous sommes immédiatement déconnectés.

Les messages suivants sont affichés par la fenêtre de sortie :

```
connexion réussie
fermeture de la connexion
```

Il serait intéressant de pouvoir parler au serveur de socket. Pour cela nous modifions les lignes PHP suivantes :

```
#!/usr/local/bin/php -q
<?php

set_time_limit(0);

$adresse = "localhost";
$port = "10000";

$connexion = socket_create (AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind ($connexion, $adresse, $port);

socket_listen ($connexion, 1);

echo "Le serveur de socket est en route !";

$client = socket_accept($connexion);
```

```
$messageEntrant = socket_read ($client, 1024);  
$messageSortie = "Vous avez dit : ".$messageEntrant."\r\n";  
socket_write ($client, $messageSortie);  
socket_close ($client);  
socket_close ($connexion);  
?>
```

Afin de prendre en considération les modifications, nous redémarrons le serveur de socket. Désormais, celui-ci est en attente d'un message du client.

Nous allons envoyer une simple chaîne de caractères à l'aide de la méthode `send` de l'objet `XMLSocket` :

```
// envoie d'une chaîne de caractère au serveur  
connexion.send ("il y'a quelqu'un ?");
```

En observant la fenêtre de sortie nous pouvons voir les messages suivants :

```
connexion réussie  
réception des données  
fermeture de la connexion
```

Aussitôt la méthode `send` exécutée, le lecteur Flash transmet au serveur de socket la chaîne de caractère spécifiée en terminant le message par un octet nul.

---

Les données transmises par le serveur sont aussi terminées par un octet nul, ce marqueur permet au lecteur Flash de connaître en interne la fin du paquet transmis.

---

Comme son nom l'indique, la classe `XMLSocket` est prévue pour échanger des données au format XML. Pourtant, nous échangeons ici une chaîne de caractère traditionnelle.

Le format XML peut être utilisé si vous souhaitez échanger des données structurées, dans d'autres cas, des simples chaînes de caractères peuvent être utilisées.

Afin de lire les données provenant du serveur, nous utilisons la l'événement `DataEvent.DATA`. L'objet événementiel diffusé possède une propriété `data` contenant les données transmises par le serveur :

```
// création d'une instance de XMLSocket  
var connexion:XMLSocket = new XMLSocket("localhost", 10000);  
  
// envoie d'une chaîne de caractère au serveur  
connexion.send ("il y'a quelqu'un ?");
```

```
// écoute des événements
connexion.addListener ( Event.CONNECT, connexionReussie );
connexion.addListener ( Event.CLOSE, fermetureConnexion );
connexion.addListener ( DataEvent.DATA, receptionDonnees );

function connexionReussie ( pEvt:Event ):void
{
    trace("connexion réussie");
}

function fermetureConnexion ( pEvt:Event ):void
{
    trace("fermeture de la connexion");
}

function receptionDonnees ( pEvt:DataEvent ):void
{
    // affiche : Vous avez dit : il y'a quelqu'un ?
    trace ( pEvt.data );
}
```

Attention, la méthode `send` n'est pas bloquante, nous devons garder à l'esprit le caractère asynchrone des échanges réalisés par une connexion socket. Seul l'événement `DataEvent.DATA` nous permet de récupérer les données transmises par le serveur et de déterminer à quel moment celles-ci sont disponibles.

Nous allons aller plus loin en développant un t'chat multi-utilisateur. Le serveur de socket XML que nous allons développer servira de relais afin de faire transiter les messages des clients connectés.

## A retenir

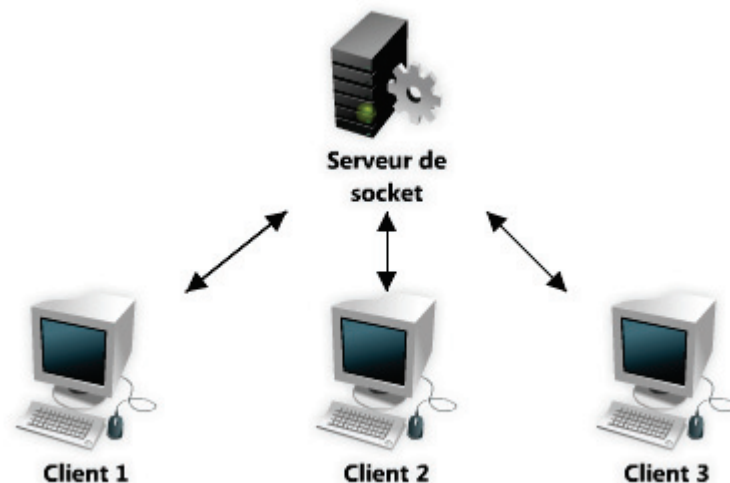
- La classe `XMLSocket` permet la connexion auprès d'un serveur de socket.
- Les données échangées entre les deux acteurs doivent obligatoirement être au format XML ou texte.
- L'objet `XMLSocket` diffuse différents événements permettant d'indiquer l'état de la connexion et du transfert des données.

## Créer un t'chat multi-utilisateur

Afin de développer notre t'chat multi utilisateur, nous allons réutiliser le moteur d'émoticones développé au cours du chapitre 16 intitulé *Le texte*.

Nous retrouvons à nouveau l'intérêt de la programmation orientée objet en facilitant la réutilisation d'objets prédéfinis.

La figure 18-3 illustre le fonctionnement d'un chat multi utilisateur, comme nous l'avons vu précédemment, le serveur agit comme relais entre les différents clients connectés :



*Figure 18-3. T'chat multi-utilisateurs.*

Au sein d'un fichier nommé `ServeurMessagerieXML.php` nous définissons le code suivant :

```
#!/usr/bin/php -q
<?php

set_time_limit(0);

$adresse = 'localhost';
$port = 10000;

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
socket_set_option($connexion, SOL_SOCKET, SO_REUSEADDR, 1);
$ret = socket_bind($connexion, $adresse, $port);
$ret = socket_listen($connexion, 5);
$tabSockets = array($connexion);

while (true)
{
    $sockets = $tabSockets;

    socket_select($sockets, $write = NULL, $except = NULL, NULL);

    foreach($sockets as $socket)
    {
```

```
if ($socket == $connexion)
{

    if (($client = socket_accept($connexion)) < 0) continue;

    else array_push($tabSockets, $client);

} else
{

    $flux = socket_recv($socket, $buffer, 2048, 0);

    if ($flux == 0)
    {

        $index = array_search($socket, $tabSockets);
        unset($tabSockets[$index]);
        socket_close($socket);

    }else
    {

        $allclients = $tabSockets;
        array_shift($allclients);
        send_Message($allclients, $socket, $buffer);

    }

}

}

}

function send_Message($clients, $socket, $donnees)
{

    foreach($clients as $client)
    {

        socket_write($client, $donnees);

    }

}

?>
```

Nous ne démarrons pas le serveur pour le moment, nous allons développer à présent la partie client de l'application de messagerie instantanée.

Lors du chapitre 16, nous avons développé une classe `MoteurEmoticone`, celle-ci va nous permettre d'afficher le texte de la conversation. Grâce à ses fonctionnalités, les utilisateurs auront la possibilité d'utiliser un ensemble d'émoticones prédéfinis.

Nous ajoutons les lignes suivantes à la classe `MoteurEmoticone` :

```
package org.bytearray.emoticones

{

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.KeyboardEvent;
    import flash.geom.Rectangle;
    import flash.text.TextField;
    import flash.ui.Keyboard;
    import org.bytearray.evenements.EvenementSaisie;
    import org.bytearray.evenements.EvenementMessage;

    public class MoteurEmoticone extends Sprite

    {

        public var contenuHistorique:TextField;
        public var chaineHistorique:String;
        public var contenuSaisie:TextField;
        public var codesTouches:Array;
        public var tableauSmileys:Array;
        public var lngTouches:int;
        public var coordonnees:Rectangle;
        public var emoticone:Emoticone;

        public function MoteurEmoticone ()

        {

            chaineHistorique = new String();

            codesTouches = new Array (":",":D",":(",":;)",":p");

            tableauSmileys = new Array ();

            lngTouches = codesTouches.length;

            addEventListener ( KeyboardEvent.KEY_DOWN, envoiMessage );

            contenuHistorique.addEventListener ( Event.SCROLL,
saisieUtilisateur );

        }

        private function envoiMessage ( pEvt:KeyboardEvent ):void

        {

            if ( pEvt.keyCode == Keyboard.ENTER )

            {

                dispatchEvent ( new EvenementSaisie (
EvenementSaisie.ENVOI_MESSAGE, contenuSaisie.text ) );

                contenuSaisie.htmlText = "";

            }

        }

    }

}
```

```
public function afficheMessage ( pEvt:EvenementMessage ):void
{
    chaineHistorique += pEvt.message;

    contenuHistorique.text = chaineHistorique;

    contenuHistorique.dispatchEvent ( new Event ( Event.SCROLL ) );
}

private function saisieUtilisateur ( pEvt:Event ):void
{
    var i:int;
    var j:int;

    var nombreSmileys:int = tableauSmileys.length;

    for ( i = 0; i< nombreSmileys; i++ ) removeChild (
tableauSmileys[i] );

    tableauSmileys = new Array();

    for ( i = 0; i<lngTouches; i++ )
    {
        j = pEvt.target.text.indexOf ( codesTouches[i] );

        while ( j!= -1 )
        {
            coordonnees = pEvt.target.getCharBoundaries ( j );

            if ( coordonnees != null ) tableauSmileys.push (
ajouteSmiley ( coordonnees, i ) );

            j = pEvt.target.text.indexOf ( codesTouches[i], j+1 );
        }
    }

    private function ajouteSmiley ( pRectangle:Rectangle, pIndex:int
):Emoticone
    {
        emoticone = new Emoticone();
        emoticone.gotoAndStop ( pIndex + 1 );

        emoticone.x = pRectangle.x + 1;
        emoticone.y = pRectangle.y - ((contenuHistorique.scrollV -
1)*(contenuHistorique.textHeight/contenuHistorique.numLines))+1;

        addChild ( emoticone );

        return emoticone;
    }
}
```



```
    }  
  }  
}
```

La classe `MoteurEmoticone` est liée à un clip contenant deux champs texte `contenuHistorique` et `contenuSaisie`.

Afin de pouvoir informer le reste de l'application de la saisie utilisateur, la classe `MoteurEmoticone` diffuse un événement `EvenementSaisie.ENVOI_MESSAGE`.

Au sein du paquetage `org.bytearray.events` nous définissons la classe `EvenementSaisie` suivante :

```
package org.bytearray.events  
  
{  
    import flash.events.Event;  
  
    public class EvenementSaisie extends Event  
    {  
        public static const ENVOI_MESSAGE:String = "envoiMessage";  
        public var saisie:String;  
  
        public function EvenementSaisie ( pType:String, pSaisie:String )  
        {  
            super( pType, false, false );  
            saisie = pSaisie;  
        }  
  
        public override function clone ():Event  
        {  
            return new EvenementSaisie ( type, saisie );  
        }  
  
        public override function toString ():String  
        {  
            return '[EvenementSaisie type="'+ type +'" bubbles=' + bubbles +  
            ' eventPhase='+ eventPhase + ' cancelable=' + cancelable + ' saisie=' + saisie  
            +']';  
        }  
    }  
}
```

```
}
```

La classe `EvenementMessage` permet d'informer l'application de l'arrivée de nouveaux messages.

Au sein du même paquetage que la classe `EvenementSaisie` nous définissons la classe `EvenementMessage` suivante :

```
package org.bytearray.evenements

{

    import flash.events.Event;

    public class EvenementMessage extends Event

    {

        public static const RECEPTION_MESSAGE:String = "receptionMessage";
        public var message:String;

        public function EvenementMessage ( pType:String, pMessage:String )

        {

            super( pType, false, false );

            message = pMessage;

        }

        public override function clone ():Event

        {

            return new EvenementMessage ( type, message );

        }

        public override function toString ():String

        {

            return '[EvenementMessage type="'+ type +' " bubbles=' + bubbles +
            ' eventPhase='+ eventPhase + ' cancelable=' + cancelable + ' message=' +
            message + ']';

        }

    }

}
```

Afin de tester la classe `MoteurEmoticone` nous associons la classe de document suivante à un nouveau document Flash CS3 :

```
package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefault;
```

```

import org.bytearray.emoticones.MoteurEmoticone;

public class Document extends ApplicationDefault
{
    public var client:MoteurEmoticone;

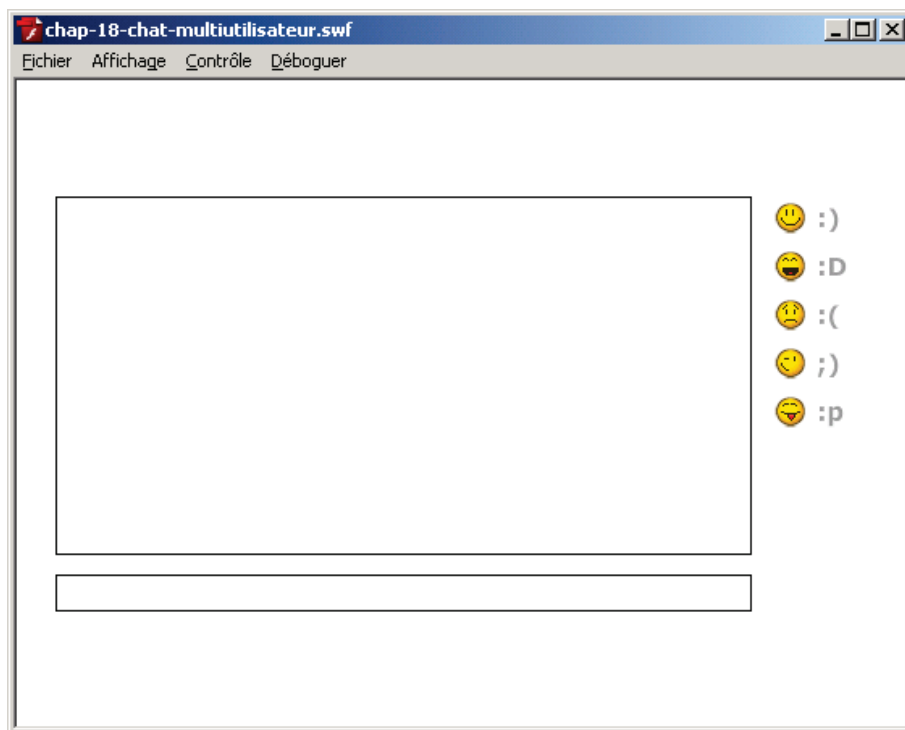
    public function Document ()
    {
        client = new MoteurEmoticone();

        client.x = (stage.stageWidth - client.width) >> 1;
        client.y = (stage.stageHeight - client.height) >> 1;

        addChild ( client );
    }
}

```

La figure 18-4 illustre l'interface de t'chat :



*Figure 18-4. Client chat multi-utilisateur.*

L'objet `MoteurEmoticone` diffuse un événement `EvenementSaisie.ENVOI_MESSAGE` que nous écoutons afin de récupérer le texte saisi :

```

package org.bytearray.document

```

```
{

import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.emoticones.MoteurEmoticone;
import org.bytearray.evenements.EvenementSaisie;

public class Document extends ApplicationDefaut
{

    public var client:MoteurEmoticone;

    public function Document ()

    {

        client = new MoteurEmoticone();
        client.x = (stage.stageWidth - client.width) >> 1;
        client.y = (stage.stageHeight - client.height) >> 1;

        addChild ( client );

        client.addEventListener ( EvenementSaisie.ENVOI_MESSAGE,
envoiMessage );

    }

    private function envoiMessage ( pEvt:EvenementSaisie ):void

    {

        // affiche : [EvenementSaisie type="envoiMessage" bubbles=false
eventPhase=2 cancelable=false saisie=salut !]
        trace( pEvt );

    }

}

}
```

Puis nous envoyons les données au serveur de socket grâce à la méthode `send` de l'objet `XMLSocket` :

```
package org.bytearray.document

{

import flash.events.DataEvent;
import flash.net.XMLSocket;
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.emoticones.MoteurEmoticone;
import org.bytearray.evenements.EvenementSaisie;

public class Document extends ApplicationDefaut
{

    private static const ADRESSE:String = "localhost";
    private static const PORT:int = 10000;

    public var client:MoteurEmoticone;
    public var connexionSocket:XMLSocket;
```

```
public function Document ()
{
    client = new MoteurEmoticone();
    client.x = (stage.stageWidth - client.width) >> 1;
    client.y = (stage.stageHeight - client.height) >> 1;

    addChild ( client );

    client.addEventListener ( EvenementSaisie.ENVOI_MESSAGE,
envoiMessage );

    connexionSocket = new XMLSocket ( Document.ADRESSE, Document.PORT
);

    connexionSocket.addEventListener ( DataEvent.DATA,
receptionDonnees );

}

private function envoiMessage ( pEvt:EvenementSaisie ):void
{
    // affiche : [EvenementSaisie type="envoiMessage" bubbles=false
eventPhase=2 cancelable=false saisie=salut !]
    trace( pEvt );

    // envoi des données auprès du serveur de socket
    connexionSocket.send ( pEvt.saisie );

}

private function receptionDonnees ( pEvt:DataEvent ):void
{
    // affiche : salut !
    trace( pEvt.data );

}

}
}
```

Le principe est très simple, aussitôt les données envoyées, le serveur de socket transmet le texte saisi par un des participants à tous les clients connectés.

Pour que l'objet `MoteurEmoticone` soit averti des messages provenant du serveur et affiche la conversation, nous diffusons un événement `EvenementMessage.RECEPTION_MESSAGE` :

```
package org.bytearray.document
{
    import flash.events.DataEvent;
```

```
import flash.net.XMLSocket;
import org.bytearray.abstrait.ApplicationDefault;
import org.bytearray.emoticones.MoteurEmoticone;
import org.bytearray.evenements.EvenementMessage;
import org.bytearray.evenements.EvenementSaisie;

public class Document extends ApplicationDefault
{

    private static const ADRESSE:String = "localhost";
    private static const PORT:int = 10000;

    public var client:MoteurEmoticone;
    public var connexionSocket:XMLSocket;

    public function Document ()

    {

        client = new MoteurEmoticone();
        client.x = (stage.stageWidth - client.width) >> 1;
        client.y = (stage.stageHeight - client.height) >> 1;

        addChild ( client );

        client.addEventListener ( EvenementSaisie.ENVOI_MESSAGE,
envoiMessage );

        // écoute de l'événement EvenementMessage.RECEPTION_MESSAGE afin
d'afficher le texte
        addEventListener ( EvenementMessage.RECEPTION_MESSAGE,
client.afficheMessage );

        connexionSocket = new XMLSocket ( Document.ADRESSE, Document.PORT
);

        connexionSocket.addEventListener ( DataEvent.DATA,
receptionDonnees );

    }

    private function envoiMessage ( pEvt:EvenementSaisie ):void

    {

        // affiche : [EvenementSaisie type="envoiMessage" bubbles=false
eventPhase=2 cancelable=false saisie=salut !]
        trace( pEvt );

        // envoi des données auprès du serveur de socket
        connexionSocket.send ( pEvt.saisie );

    }

    private function receptionDonnees ( pEvt:DataEvent ):void

    {

        // diffusion d'un événement EvenementMessage.RECEPTION_MESSAGE
afin d'afficher les données reçues du serveur
        dispatchEvent ( new EvenementMessage (
EvenementMessage.RECEPTION_MESSAGE, pEvt.data ) );

    }

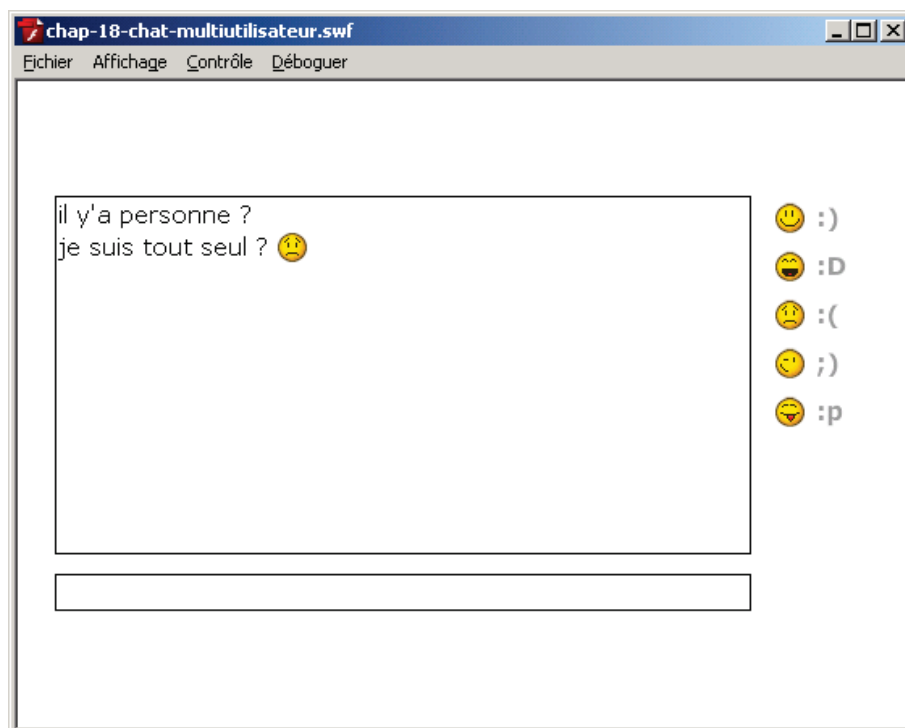
}
```

```

    }
}
}

```

En testant notre client nous remarquons que le texte est bien affiché lorsque nous l’envoyons, comme l’illustre la figure 18-5 :



*Figure 18-5. Client flash connecté.*

Si nous connectons un deuxième client, notre chat multi utilisateur fonctionne. Vous pouvez à présent déployer le serveur de socket ainsi que le client Flash sur votre serveur et diffuser l’adresse aux personnes avec qui vous souhaitez discuter.

La classe `XMLSocket` permet le développement d’applications temps réel de manière très souple. Bien entendu, de nombreux serveurs existent déjà et sont conçus pour fonctionner avec Flash à l’aide de la classe `XMLSocket`.

Parmi ceux là nous pouvons citer les serveurs suivants :

- Unity : serveur de socket XML payant de Colin Moock - <http://www.moock.org/unity>
- SmartFox Server : serveur de socket XML payant - <http://www.smartfoxserver.com>

- Jabber : serveur de socket XML gratuit destiné à la création d'applications de messageries instantanées - <http://www.jabber.org>

Une des limitations de la classe `XMLSocket` est liée à limitation du format des données échangées.

Nous allons voir dans cette nouvelle partie dans quelle mesure la classe `Socket` peut s'avérer utile.

### A retenir

- La classe `XMLSocket` permet d'échanger des données au format XML et texte entre un client et un serveur de socket.
- En tant que relais, le serveur de socket permet de faire transiter des informations entre plusieurs clients connectés.
- Chaque message est terminé par un octet nul.

## La classe Socket

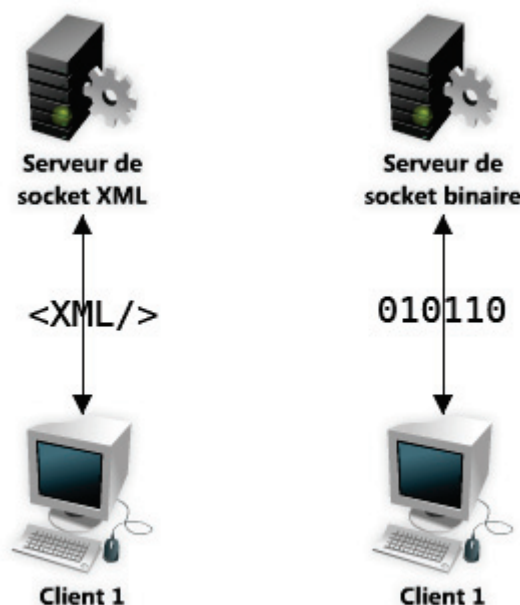
ActionScript 3 intègre une nouvelle classe `Socket` se différenciant quelque peu de la classe `XMLSocket`.

Comme nous l'avons vu précédemment, la classe `XMLSocket` transmet chaque message en le terminant par un octet nul. Ce comportement provoque un troncage des données binaire.

La classe `Socket` ne souffre pas de cette limitation et offre la possibilité de transférer un flux binaire brut sans altérations.

Si nous devons schématiser le fonctionnement d'un serveur de socket binaire nous pourrions l'illustrer de la manière suivante :





*Figure 18-6. XMLSocket et Socket.*

Nous allons ensemble développer un serveur de socket binaire afin de transférer par une connexion socket un élément graphique telle une image ou un SWF.

Outre l'expérimentation technique, ce procédé permet d'éviter la mise en cache des éléments graphiques au sein du navigateur. L'élément graphique est chargé directement en mémoire par le lecteur Flash puis affiché, cette approche évite la mise en cache de fichiers SWF et rend leur décompilation difficile.

## A retenir

- La classe `Socket` permet d'échanger des données au format binaire brutes.
- Grâce au caractère bas niveau de la classe `Socket`, il est possible d'implémenter n'importe quel protocole d'application.

## Créer un serveur de socket binaire

Nous allons à nouveau utiliser PHP afin de créer notre serveur de socket binaire. L'intérêt de ce dernier sera de pouvoir recevoir des requêtes et d'y répondre. Nous allons ainsi définir notre propre protocole afin de normaliser les échanges entre les deux acteurs.

Dans un fichier PHP nommé `ServeurSocket.php` nous définissons le code suivant :

```
#!/usr/bin/php -q
<?php

$adresse = "localhost";

$port = "10000";

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind($connexion, $adresse, $port);

socket_listen($connexion, 1);

echo "Le serveur de socket binaire est en route !";

$client = socket_accept($connexion);

socket_close($client);
socket_close($connexion);

?>
```

Une fois le serveur démarré nous pouvons nous y connecter de la manière suivante :

```
// création de la connexion
var connexion:Socket = new Socket();

// connexion au serveur de socket
connexion.connect( "localhost", 10000 );

// écoute des différents événements
connexion.addEventListener( Event.CONNECT, clientConnecte );
connexion.addEventListener( Event.CLOSE, clientDeconnecte );

function clientConnecte ( pEvt:Event ):void
{
    trace("client connecté");
}

function clientDeconnecte ( pEvt:Event ):void
{
    trace("client déconnecté");
}
```

Si nous testons le code précédent, nous remarquons que la connexion est aussitôt fermée.

Le panneau de sortie affiche les messages suivants :

```
client connecté
client déconnecté
```

Nous modifions le code du serveur de socket binaire en ajoutant un appel à la méthode `socket_write` pour envoyer les données au client en cours :

```
#!/usr/bin/php -q
<?php

$adresse = "localhost";

$port = "10000";

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind($connexion, $adresse, $port);

socket_listen($connexion, 1);

echo "Le serveur de socket binaire est en route !";

$client = socket_accept($connexion);

$messageSortant = "Bienvenue sur le serveur de socket, mais désolé vous êtes
déconnecté !";

socket_write ($client, $messageSortant );

socket_close($client);
socket_close($connexion);

?>
```

Nous envoyons désormais des données depuis le serveur avant de clore la connexion.

Lorsque le serveur transmet les données, l'événement

`ProgressEvent.SOCKET_DATA` est diffusé :

```
// création de la connexion
var connexion:Socket = new Socket();

// connexion au serveur de socket
connexion.connect( "localhost", 10000 );

// écoute des différents événements
connexion.addEventListener( Event.CONNECT, clientConnecte );
connexion.addEventListener( ProgressEvent.SOCKET_DATA, donneesRecues );
connexion.addEventListener( Event.CLOSE, clientDeconnecte );

function clientConnecte ( pEvt:Event ):void

{

    trace("client connecté");

}

function donneesRecues ( pEvt:ProgressEvent ):void

{
```

```
        trace("données reçues");
    }

    function clientDeconnecte ( pEvt:Event ):void
    {
        trace("client déconnecté");
    }
}
```

Si nous testons le code précédent nous voyons que le client parvient à se connecter, puis les données sont reçues, enfin le client est automatiquement déconnecté.

Le panneau de sortie affiche les messages suivants :

```
client connecté
données reçues
client déconnecté
```

Contrairement à la classe `XMLSocket`, les données reçues par la classe `Socket` doivent être décodées.

Afin de lire les données transmises par le serveur nous utilisons ici la méthode `readUTFBytes` de l'objet `Socket` dont voici la signature :

```
public function readUTFBytes(length:uint):String
```

Celle-ci accepte un paramètre `length` correspondant au nombre d'octets à lire et à renvoyer sous forme de chaîne de caractères UTF-8.

Afin de lire les données disponibles, nous utilisons la propriété `bytesAvailable` :

```
function donneesRecues ( pEvt:ProgressEvent ):void
{
    // affiche : Bienvenue sur le serveur de socket, mais désolé vous êtes
    // déconnecté !
    trace( pEvt.target.readUTFBytes ( pEvt.target.bytesAvailable ) );
}
```

N'oubliez pas que dans un contexte de transfert de données TCP/IP, les données arrivent par paquet d'octets. Il est donc nécessaire de lire le flux de données au fur et à mesure que les données arrivent au sein du lecteur. Celles-ci s'accumulent au sein de l'objet `Socket` et peuvent être lues à l'aide des différentes méthodes définies par la classe `Socket`.

La propriété `bytesAvailable` nous permet de récupérer le nombre d'octets disponibles actuellement au sein du flux téléchargé et évite de sortir du flux de données.

Si nous tentons de lire des octets non disponibles, l'erreur suivante est levée à l'exécution :

```
| Error: Error #2030: Fin de fichier détectée.
```

Nous reviendrons en détail sur les différentes méthodes de lecture et d'écriture de flux binaire au cours du chapitre 20 intitulé *ByteArray*.

## A retenir

- Contrairement à la classe `XMLSocket`, la classe `Socket` reçoit un flux binaire brut de la part du serveur.
- Ces données doivent être décodées à l'aide des différentes méthodes de la classe `Socket`.

## Echanger des données

Nous avons évoqué en début de chapitre la notion de protocole d'application afin d'échanger des messages entre un client un serveur.

Les protocoles d'applications tels FTP, SMTP, ou POP s'appuient sur des messages prédéfinis afin d'appeler certaines fonctionnalités auprès du serveur de socket. Nous allons reproduire ce même comportement en demandant à notre serveur de socket de transférer un fichier graphique présent à ses côtés.

En passant la chaîne de caractère `"ENVOIE"` suivie du nom du fichier, le serveur renverra le flux du fichier demandé.

Afin d'intégrer ce protocole d'application nous modifions les lignes suivantes au sein du serveur de socket binaire :

```
#!/usr/bin/php -q
<?php

$adresse = "localhost";

$port = "10000";

$connexion = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);

socket_bind($connexion, $adresse, $port);

socket_listen($connexion, 1);

echo "Le serveur de socket binaire est en route !";

$client = socket_accept($connexion);

$sactif = true;

$COMMANDE_ENVOIE = "ENVOIE";

do
```

```
{  
  
    $actif = socket_read($client, 1024, PHP_BINARY_READ);  
  
    $array = split (" ", trim($actif));  
    $commande = $array[0];  
    $fichier = $array[1];  
  
    $pointeur = fopen ($fichier, "rb");  
    $donnees = fread ($pointeur, filesize ($fichier));  
    fclose ($pointeur);  
  
    if ( $commande == $COMMANDE_ENVOIE )  
    {  
  
        $longueur = pack('N', strlen($donnees));  
        socket_write($client, $longueur);  
        socket_write($client, $donnees);  
  
    } else socket_write ($client, "commande non reconnue");  
  
} while ( $actif );  
  
socket_close($client);  
socket_close($connexion);  
  
?>
```

Rappelez vous d'un point essentiel, le protocole TCP/IP transfère les données par paquets, c'est à nous de nous assurer que la totalité des données ont été transférées.

Il est donc impératif de récupérer l'ensemble des paquets cotés client avant de tenter d'afficher le fichier transféré :

```
// création de la connexion  
var connexion:Socket = new Socket();  
  
// connexion au serveur de socket  
connexion.connect( "localhost", 10000 );  
  
// écoute des différents événements  
connexion.addEventListener( Event.CONNECT, clientConnecte );  
connexion.addEventListener( ProgressEvent.SOCKET_DATA, donneesRecues );  
connexion.addEventListener( Event.CLOSE, clientDeconnecte );  
connexion.addEventListener( IOErrorEvent.IO_ERROR, erreurConnexion );  
  
// demande du chargement du fichier  
connexion.writeUTFBytes ("ENVOIE:DSC02602.JPG\r\n");  
connexion.flush();  
  
function clientConnecte ( pEvt:Event ):void  
{  
  
    trace("client connecté");  
  
}  
  
function clientDeconnecte ( pEvt:Event ):void
```

```
{
    trace("client déconnecté");
}

function erreurConnexion ( pEvt:Event ):void
{
    trace("erreur de connexion au serveur");
}

// création d'un tableau binaire pour contenir les données entrantes
var donnees:ByteArray = new ByteArray();

// nombre d'octets totaux à transférer
var longueur:Number;

function donneesRecues ( pEvt:ProgressEvent ):void
{
    // nous récupérons le nombre d'octets totaux
    if ( !longueur ) longueur = pEvt.target.readUnsignedInt();

    // les données sont progressivement stockées au sein du tableau donnees
    pEvt.target.readBytes ( donnees, donnees.length, pEvt.target.bytesAvailable
    );

    // données chargées
    if ( donnees.length == longueur ) trace ("transfert des données
    terminée");
}
```

Grâce à la méthode `writeUTFBytes`, nous encodons la chaîne de caractère passée en paramètre en flux binaire. Afin de transmettre celle-ci au serveur nous poussons les données à l'aide de la méthode `flush`.

Aussitôt la commande reçue, le serveur renvoie la longueur totale des données à recevoir, puis le flux de l'élément graphique est transmis.

Afin de sauver les données entrantes, nous créons un tableau de sauvegarde dans lequel nous plaçons les données téléchargées lors de la diffusion de l'événement `ProgressEvent.SOCKET_DATA`.

Souvenez-vous, nous avons découvert l'objet `Loader` lors du chapitre 13, ce dernier permettait le chargement de contenu externe. La classe `Loader` définit une méthode `loadBytes` permettant de rendre graphiquement un flux binaire passé en paramètre.

Voici la signature de la méthode `loadBytes` :

```
public function loadBytes(bytes:ByteArray, context:LoaderContext = null):void
```

Détail de chacun des paramètres :

- `bytes` : le flux binaire à rendre graphiquement.
- `context` : le contexte de chargement, lié au modèle de sécurité.

Attention, la méthode `loadBytes` n'accepte que des flux binaires compatibles avec l'objet `Loader`.

En d'autres termes, seuls les flux d'images ou de SWF pourront être affichés. Si un flux non compatible est passé à la méthode `loadBytes`, une erreur de type `IOErrorEvent.IO_ERROR` est levée.

Dans le code suivant, un objet `Loader` est créé. Une fois les données totalement téléchargées, nous les injectons au sein de ce dernier :

```
// création de la connexion
var connexion:Socket = new Socket();

// connexion au serveur de socket
connexion.connect( "localhost", 10000 );

// écoute des différents événements
connexion.addEventListener( Event.CONNECT, clientConnecte );
connexion.addEventListener( ProgressEvent.SOCKET_DATA, donneesRecues );
connexion.addEventListener( Event.CLOSE, clientDeconnecte );
connexion.addEventListener( IOErrorEvent.IO_ERROR, erreurConnexion );

// demande du chargement du fichier
connexion.writeUTFBytes ( "ENVOIE:DSC02602.JPG\r\n" );
connexion.flush();

function clientConnecte ( pEvt:Event ):void
{
    trace("client connecté");
}

function clientDeconnecte ( pEvt:Event ):void
{
    trace("client déconnecté");
}

function erreurConnexion ( pEvt:Event ):void
{
    trace("erreur de connexion au serveur");
}

var chargeur:Loader = new Loader();
```



```
addChild ( chargeur );

// création d'un tableau binaire pour contenir les données entrantes
var donnees:ByteArray = new ByteArray();

// nombre d'octets totaux à transférer
var longueur:Number;

function donneesRecues ( pEvt:ProgressEvent ):void
{
    // nous récupérons le nombre d'octets totaux
    if ( !longueur ) longueur = pEvt.target.readUnsignedInt();

    // les données sont progressivement stockées au sein du tableau donnees
    pEvt.target.readBytes ( donnees, donnees.length, pEvt.target.bytesAvailable
    );

    // lorsque le flux binaire est totalement chargé nous l'affichons grâce à
    l'objet Loader
    if ( donnees.length == longueur ) chargeur.loadBytes( donnees );
}
```

En testant le code précédent nous remarquons que le fichier est téléchargé par la connexion socket puis affiché au sein du lecteur.

Une fois les données injectées, l'objet `Loader` diffuse un événement `Event.COMPLETE`.

Dans le code suivant, nous redimensionnons l'image et la centrons une fois le flux affiché :

```
chargeur.contentLoaderInfo.addEventListener( Event.COMPLETE,
injectionTerminee );

function injectionTerminee ( pEvt:Event ):void
{
    var contenu:DisplayObject = pEvt.target.content;
    var objetChargeur:Loader = pEvt.target.loader;

    if ( contenu is Bitmap ) Bitmap ( contenu ).smoothing = true;

    var ratio:Number = Math.min ( 350 / pEvt.target.content.width, 350 /
pEvt.target.content.height );

    objetChargeur.scaleX = objetChargeur.scaleY = ratio;

    objetChargeur.x = (stage.stageWidth - objetChargeur.width) / 2;
    objetChargeur.y = (stage.stageHeight - objetChargeur.height) / 2;
}
```

Le résultat est illustré par la figure 18-7 :



*Figure 18-7. Image chargée par connexion socket.*

Il serait intéressant de rendre cette communication transparente au sein d'un objet spécifique. Nous allons créer une classe `ChargeurFlux` qui procèdera en interne aux différentes commandes du protocole.

Au sein d'un paquetage `org.bytearray.chargeur` nous définissons la classe `ChargeurFlux` suivante :

```
package org.bytearray.chargeur

{

    import flash.events.EventDispatcher;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;
    import flash.net.Socket;

    public class ChargeurFlux extends EventDispatcher
    {

        private var connexion:Socket;

        public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )
        {

            connexion = new Socket(pAdresse, pPort);

            connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
```

```
        connexion.addEventListener ( ProgressEvent.SOCKET_DATA,
transfertDonnees );
        connexion.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

    }

    private function redirigeEvenement ( pEvt:Event ):void
    {

        dispatchEvent ( pEvt );

    }

    private function transfertDonnees ( pEvt:ProgressEvent ):void
    {

    }

}

}
```

Puis nous ajoutons un objet **ByteArray** interne afin d'accueillir le flux téléchargé :

```
package org bytearray.chargeur
{

    import flash.events.EventDispatcher;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;
    import flash.net.Socket;
    import flash.utils.ByteArray;

    public class ChargeurFlux extends EventDispatcher
    {

        private var connexion:Socket;
        private var donnees:ByteArray;

        public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )
        {

            connexion = new Socket(pAdresse, pPort);

            donnees = new ByteArray();

            connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
            connexion.addEventListener ( ProgressEvent.SOCKET_DATA,
transfertDonnees );
            connexion.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

        }

    }
```

```
private function redirigeEvenement ( pEvt:Event ):void
{
    dispatchEvent ( pEvt );
}

private function transfertDonnees ( pEvt:ProgressEvent ):void
{
}

}

}
```

Puis nous implémentons les méthodes **charge** et **connecte** :

```
package org.bytearray.chargeur
{
    import flash.events.EventDispatcher;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;
    import flash.net.Socket;
    import flash.utils.ByteArray;

    public class ChargeurFlux extends EventDispatcher
    {
        private var connexion:Socket;
        private var donnees:ByteArray;

        private static const ENVOIE:String = "ENVOIE";

        public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )
        {
            connexion = new Socket(pAdresse, pPort);
            donnees = new ByteArray();

            connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
            connexion.addEventListener ( ProgressEvent.SOCKET_DATA,
transfertDonnees );
            connexion.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );
        }

        public function charge ( pFichier:String ):void
        {

```

```
        connexion.writeUTFBytes ( ChargeurFlux.ENVOIE + " " + pFichier +
"\r\n" );
        connexion.flush();

    }

    public function connecte ( pAdresse:String, pPort:int ):void
    {

        connexion.connect ( pAdresse, pPort );

    }

    private function redirigeEvenement ( pEvt:Event ):void
    {

        dispatchEvent ( pEvt );

    }

    private function transfertDonnees ( pEvt:ProgressEvent ):void
    {

    }

}

}
```

Enfin, nous ajoutons la logique associée afin de déterminer la fin du chargement du flux :

```
package org.bytearray.chargeur
{

    import flash.events.EventDispatcher;
    import flash.events.Event;
    import flash.events.ProgressEvent;
    import flash.events.IOErrorEvent;
    import flash.net.Socket;
    import flash.utils.ByteArray;
    import org.bytearray.chargeur.events.EvenementChargeurFlux;

    public class ChargeurFlux extends EventDispatcher
    {

        private var connexion:Socket;
        private var donnees:ByteArray;
        private var longueur:Number;

        private static const ENVOIE:String = "ENVOIE";

        public function ChargeurFlux ( pAdresse:String=null, pPort:int=0 )
        {

        }

    }

}
```

```
        connexion = new Socket(pAdresse, pPort);

        donnees = new ByteArray();

        connexion.addEventListener ( Event.CONNECT, redirigeEvenement );
        connexion.addEventListener ( ProgressEvent.SOCKET_DATA,
transfertDonnees );
        connexion.addEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

    }

    public function charge ( pFichier:String ):void
    {

        connexion.writeUTFBytes ( ChargeurFlux.ENVOIE + " " + pFichier +
"\r\n" );
        connexion.flush();

    }

    public function connecte ( pAdresse:String, pPort:int ):void
    {

        connexion.connect ( pAdresse, pPort );

    }

    private function redirigeEvenement ( pEvt:Event ):void
    {

        dispatchEvent ( pEvt );

    }

    private function transfertDonnees ( pEvt:ProgressEvent ):void
    {

        if ( !longueur ) longueur = pEvt.target.readUnsignedInt();

        pEvt.target.readBytes ( donnees, donnees.length,
pEvt.target.bytesAvailable );

        if ( donnees.length == longueur )
        {

            dispatchEvent ( new EvenementChargeurFlux (
EvenementChargeurFlux.TERME, donnees ) );
            donnees = new ByteArray();

        }

    }

}

}
```

Afin de charger un élément graphique stocké sur le serveur, nousinstancions simplement l'objet `ChargeurFlux` puis nous appelons sa méthode `charge` :

```
import org.bytearray.chargeur.ChargeurFlux;
import org.bytearray.chargeur.evenements.EvenementChargeurFlux;

var monChargeur:ChargeurFlux = new ChargeurFlux("localhost", 10000);
monChargeur.charge ("blason.png");

var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener( Event.COMPLETE,
injectionTerminee );

addChild ( chargeur );

monChargeur.addEventListener ( EvenementChargeurFlux.TERMINE, affiche );

function affiche ( pEvt:EvenementChargeurFlux ):void
{
    chargeur.loadBytes( pEvt.donnees );
}

function injectionTerminee ( pEvt:Event ):void
{
    var contenu:DisplayObject = pEvt.target.content;
    var objetChargeur:Loader = pEvt.target.loader;

    if ( contenu is Bitmap ) Bitmap ( contenu ).smoothing = true;

    var ratio:Number = Math.min ( 350 / pEvt.target.width, 350 /
pEvt.target.height );

    objetChargeur.scaleX = objetChargeur.scaleY = ratio;

    objetChargeur.x = (stage.stageWidth - objetChargeur.width) / 2;
    objetChargeur.y = (stage.stageHeight - objetChargeur.height) / 2;
}
```

La classe `ChargeurFlux` peut ainsi être utilisée pour le chargement de fichiers SWF afin d'éviter leur mise en cache au sein du navigateur et rendre leur décompilation moins facile :

```
| monChargeur.charge ("monsite.swf");
```

A vous d'imaginer de nouvelles fonctionnalités !

---

## A retenir

---

- La méthode `writeUTFBytes` permet d'écrire une chaîne de caractères au format binaire.
- La méthode `flush` permet d'envoyer les données binaires au serveur de socket.

Dans le prochain chapitre nous découvrirons la technologie Flash Remoting afin de dialoguer de manière optimisée avec différents langages serveurs et bases de données.



# 19

## Flash Remoting

<b>LA TECHNOLOGIE.....</b>	<b>1</b>
UN FORMAT OPTIMISÉ.....	2
<b>PASSERELLE REMOTING.....</b>	<b>4</b>
DÉPLOIEMENT.....	5
<b>LE SERVICE.....</b>	<b>7</b>
SE CONNECTER AU SERVICE .....	15
LA CLASSE RESPONDER .....	16
APPELER UNE MÉTHODE DISTANTE .....	18
ÉCHANGER DES DONNEES PRIMITIVES.....	22
ÉCHANGER DES DONNEES COMPOSITES.....	27
ÉCHANGER DES DONNEES TYPEES.....	32
ENVOYER UN EMAIL AVEC FLASH REMOTING.....	33
EXPORTER UNE IMAGE .....	40
SE CONNECTER A UNE BASE DE DONNEES.....	49
<b>SECURITE .....</b>	<b>61</b>
<b>LA CLASSE SERVICE.....</b>	<b>62</b>

### La technologie

Développé à l'origine par Macromedia, *Flash Remoting* est une technologie visant à optimiser grandement les échanges client-serveur.

Le lecteur Flash 6 fut le premier à en bénéficier, mais la puissance de celle-ci ne fut pas perçue immédiatement par la communauté due en partie à un manque de documentation et d'informations à ce sujet. Grâce à Flash Remoting nous allons optimiser notre temps de développement d'applications dynamiques et apprendre à penser différemment nos échanges client-serveur. Des frameworks tels Flex

ou AIR s'appuient aujourd'hui fortement sur Flash Remoting trop souvent inconnu des développeurs Flash.

Nous allons découvrir au sein de ce chapitre comment tirer profit de cette technologie en ActionScript 3 à travers différentes applications.

## A retenir

- La technologie Flash Remoting vu le jour en 2001 au sein du lecteur 6 (Flash MX).
- Flash Remoting permet d'optimiser les échanges client-serveur.

## Un format optimisé

Toute la puissance de Flash Remoting réside dans le format d'échanges utilisé, connu sous le nom d'AMF (*Action Message Format*).

Afin de bien comprendre l'intérêt de ce format, il convient de revenir sur le fonctionnement interne du lecteur Flash dans un contexte de chargement et d'envoi de données.

Nous avons vu au cours du chapitre 14 intitulé *Chargement et envoi de données* que les données échangées entre le lecteur Flash et le script serveur étaient par défaut réalisées au format texte.

Dans le cas d'échanges de variables encodées URL, une représentation texte des variables doit être réalisée. Nous devons donc nous assurer manuellement de la sérialisation et désérialisation des données ce qui peut s'avérer long et fastidieux, surtout dans un contexte de large flux de données.

Nous pouvons alors utiliser le format XML, mais là encore bien que l'introduction de la norme ECMAScript pour XML (E4X) ait sensiblement amélioré les performances d'interprétation de flux XML, dans un contexte d'échanges, le flux XML reste transporté au format texte ce qui n'est pas optimisé en terme de bande passante.

Grâce au format AMF, le lecteur Flash se charge de sérialiser et désérialiser les données ActionScript nativement. Nous pouvons ainsi échanger des données complexes typées, sans se soucier de la manière dont cela est réalisé.

Lorsque nous souhaitons transmettre des données par Flash Remoting, le lecteur encode un paquet AMF binaire compressé contenant les données sérialisées, puis le transmet au script serveur par la méthode POST par le biais du protocole HTTP.

Voici un extrait de la transaction HTTP :

```
POST gateway.php HTTP/1.1
Host: www.bytearray.org
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.1.11)
Gecko/20071127 Firefox/2.0.0.11
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0
.8,image/png,*/*;q=0.5
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.bytearray.org/wp-
content/projects/jpegencoder/media_snapshot.swf
Content-type: application/x-amf
Content-length: 4944
Paquet AMF
```

Une fois le paquet AMF décodé par le serveur, ce dernier répond au lecteur Flash en lui transmettant un nouveau paquet AMF de réponse :

```
HTTP/1.1 200 OK
Date: Sat, 02 Feb 2008 02:55:40 GMT
Server: Apache/2.0.54 (Debian GNU/Linux) PHP/4.3.10-22 mod_ssl/2.0.54
OpenSSL/0.9.7e mod_perl/1.999.21 Perl/v5.8.4
X-Powered-By: PHP/4.3.10-22
Expires: Sat, 2 Feb 2008 03:55:40 GMT
Cache-Control: no-store
Pragma: no-store
Content-length: 114
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: application/x-amf
Paquet AMF
```

Il est important de comprendre que contrairement au format texte, le format AMF est un format binaire compressé natif du lecteur Flash.

L'utilisation du format AMF nous permet donc d'optimiser la bande passante utilisée et d'augmenter les performances d'une application en s'affranchissant totalement de toute sérialisation manuelle des données.

James Ward a dernièrement publié une application comparant les performances d'échanges au format JSON, XML et AMF.

L'application est disponible à l'adresse suivante :

<http://www.jamesward.org/blazebench/>

Bien que le lecteur Flash puisse nativement encoder ou décoder un paquet AMF, l'application serveur se doit elle aussi de pouvoir décoder ce même paquet.

C'est ici qu'intervient la notion de *passerelle remoting*.

---

## A retenir

---

- L'acronyme AMF signifie *Action Message Format*.
- Flash Remoting est basé sur un échange des données au format AMF binaire par le biais du protocole HTTP.
- Encoder et décoder un paquet AMF est beaucoup plus rapide et moins gourmand qu'une sérialisation et désérialisation au format texte.
- Deux versions du format AMF existent aujourd'hui : AMF0 (ActionScript 1 et 2) et AMF3 (ActionScript 3).
- Les formats AMF0 et AMF3 sont ouverts et documentés.
- Aujourd'hui, le format AMF est utilisé au sein des classes `NetConnection`, `LocalConnection`, `SharedObject`, `ByteArray`, `Socket` et `URLStream`.

## Passerelle remoting

Lorsque le lecteur Flash transmet un paquet AMF à un script serveur, celui-ci n'est pas par défaut en mesure de le décoder.

Des projets appelés passerelles remoting ont ainsi vu le jour, permettant à des langages serveurs tels PHP, Java ou C# de comprendre le format AMF. Chaque passerelle est ainsi liée à un langage serveur et se charge de convertir le flux AMF transmis en données compatibles.

Afin de développer ces passerelles, le format AMF a dû être piraté par la communauté afin de comprendre comment le décoder. Initialement non documenté, Adobe a finalement rendu public les spécifications du format AMF en décembre 2007. Toute personne souhaitant développer une passerelle pour un langage spécifique peut donc se baser sur ces spécifications fournies par Adobe.

Celles-ci peuvent être téléchargées à l'adresse suivante :

<http://labs.adobe.com/technologies/blazeds/>

En plus d'ouvrir le format AMF, Adobe a décidé de fournir une passerelle officielle pour la plateforme Java J2EE nommée *BlazeDS* téléchargeable à la même adresse.

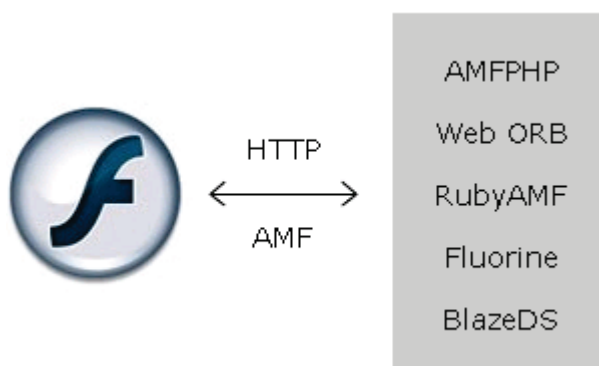
Il existe aujourd'hui un grand nombre de passerelles remoting pour la plupart open source, le tableau suivant regroupe les plus connues associées à chaque langage :

Nom	Langage	Open Source	Lien
-----	---------	-------------	------

AMFPHP	PHP	Oui	<a href="http://www.amfphp.org">http://www.amfphp.org</a>
Web ORB	PHP, .NET, Ruby, Java.	Non	<a href="http://www.themidnightcoders.com">http://www.themidnightcoders.com</a>
BlazeDS	Java	Oui	<a href="http://labs.adobe.com/technologies/blazeds/">http://labs.adobe.com/technologies/blazeds/</a>
Ruby AMF	Ruby	Oui	<a href="http://www.rubyamf.org">http://www.rubyamf.org</a>
Fluorine	.NET	Oui	<a href="http://fluorine.thesilentgroup.com">http://fluorine.thesilentgroup.com</a>

*Tableau 1. Passerelles AMF.*

La figure 19-1 illustre le modèle de communication entre le lecteur Flash et la passerelle remoting :



*Figure 19-1. Communication entre le lecteur Flash et la passerelle remoting.*

Les paquets AMF devant forcément transiter par la passerelle afin d’être décodés, le lecteur ne se connecte donc *jamais* directement auprès du script serveur, mais auprès de la passerelle.

Nous allons utiliser tout au long du chapitre, la passerelle remoting AMFPHP afin de simplifier nos échanges avec le serveur.

## Déploiement

Avant de déployer AMFPHP il convient de télécharger le projet à l’adresse suivante :

<http://sourceforge.net/projects/amfphp>

Attention, le format AMF3 n'est pris en charge que depuis la version 1.9, nous veillerons donc à télécharger une version égale ou ultérieure.

---

Souvenez-vous, ActionScript 3 utilise le format AMF3, les anciennes versions d'ActionScript utilisent le format AMF0.

---

Une fois AMFPHP téléchargé, nous extrayons l'archive et obtenons les répertoires illustrés par la figure 19-2 :

Nom	Taille	Type
browser		Dossier de fichiers
core		Dossier de fichiers
services		Dossier de fichiers
.htaccess	1 Ko	Fichier HTACCESS
gateway.php	6 Ko	OpenStudio File
globals.php	1 Ko	OpenStudio File
json.php	1 Ko	OpenStudio File
phpinfo.php	1 Ko	OpenStudio File
xmlrpc.php	1 Ko	OpenStudio File

*Figure 19-2. Fichiers AMFPHP.*

Voici le détail des trois répertoires ainsi que du fichier `gateway.php` :

- `browser` : contient une application de débogage que nous allons découvrir très bientôt.
- `core` : il s'agit des sources d'AMFPHP. Dans la majorité des cas nous n'irons jamais au sein de ce répertoire.
- `services` : les services distants sont placés au sein de ce répertoire. C'est le répertoire le plus important pour nous.
- `gateway.php` : la passerelle à laquelle nous nous connectons depuis Flash.

Une des forces des différentes passerelles remoting réside dans leur simplicité de déploiement sur le serveur.

Les passerelles sont déployables sur des serveurs mutualisés et ne nécessitent aucun accès privilégié au niveau de l'administration du serveur. AMFPHP requiert une version PHP 4.3.0 ou supérieure, l'utilisation de PHP 5 ajoutant quelques fonctionnalités intéressantes à AMFPHP.

Nous plaçons le répertoire `amfphp` contenant les fichiers illustrés précédemment sur notre serveur au sein d'un répertoire `echanges` et accédons à l'adresse suivante pour vérifier que tout fonctionne correctement :

<http://localhost/echanges/gateway.php>

Si tout fonctionne, nous obtenons le message suivant :

```
amfphp and this gateway are installed correctly. You may now connect to this gateway from Flash.
```

```
Note: If you're reading an old tutorial, it will tell you that you should see a download window instead of this message. This confused people so this is the new behaviour starting from amfphp 1.2.
```

Si vous souhaitez utiliser une autre passerelle remoting que AMFPHP, soyez rassurés, le fonctionnement de la plupart d'entre elles est similaire.

## A retenir

- La passerelle remoting joue le rôle d'intermédiaire entre le lecteur Flash et l'application serveur.
- Le déploiement d'une passerelle remoting ne prend que quelques secondes.
- Aucun droit spécifique n'est nécessaire auprès du serveur. Il est donc possible d'utiliser Flash Remoting sur un serveur mutualisé.
- AMFPHP requiert une version de PHP 4.3.0 au minimum.

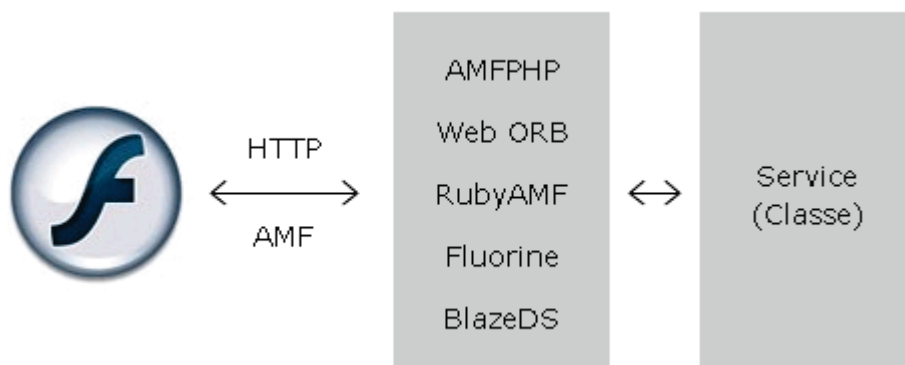
## Le service

Contrairement aux moyens de communication étudiés précédemment, Flash Remoting définit la notion de *service*. Le service est une classe définissant les méthodes que nous souhaitons appeler depuis l'application Flash.

Nous pourrions ainsi appeler le service "classe distante".

Un des avantages de Flash Remoting est lié à sa conception orientée objet. Au lieu d'appeler différents scripts serveurs stockés au sein de plusieurs fichiers PHP, nous appelons directement depuis Flash des méthodes définies au sein du service.

La figure 19-3 illustre l'idée :



*Figure 19-3. Service distant associé à la passerelle.*

Nous allons créer notre premier service en plaçant au sein du répertoire `services`, une classe PHP nommée `Echanges.php`.

Celle-ci est définie au sein d'un paquetage `org.bytearray.test` :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }

    function premierAppel ( )
    {
    }
}
?>
```

La classe `Echanges` doit donc être accessible à l'adresse suivante :

<http://localhost/echanges/services/org/bytearray/test/Echanges.php>

Notons que contrairement à ActionScript 3, PHP 4 et 5 n'intègrent pas de mot clé `package`, ainsi la classe ne contient aucune indication liée à son paquetage. La classe PHP `Echanges` définit une seule méthode `premierAppel` que nous allons pouvoir tester directement depuis un navigateur grâce à un outil extrêmement pratique appelé *Service Browser* (explorateur de services).

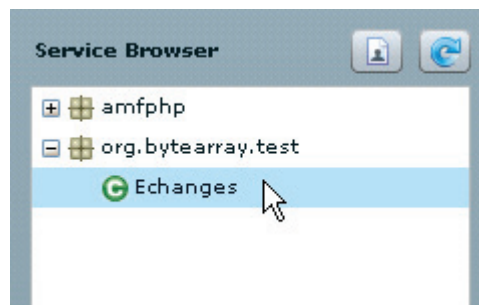
Dans notre exemple, l'explorateur de service est accessible à l'adresse suivante :

<http://localhost/echanges/browser/>

L'explorateur de service est une application Flex développée afin de pouvoir tester depuis notre navigateur les différentes méthodes de notre service. La partie gauche de l'application indique les services actuellement déployés. En déroulant les nœuds nous pouvons accéder à un service spécifique.

La figure 19-4 illustre l'explorateur :



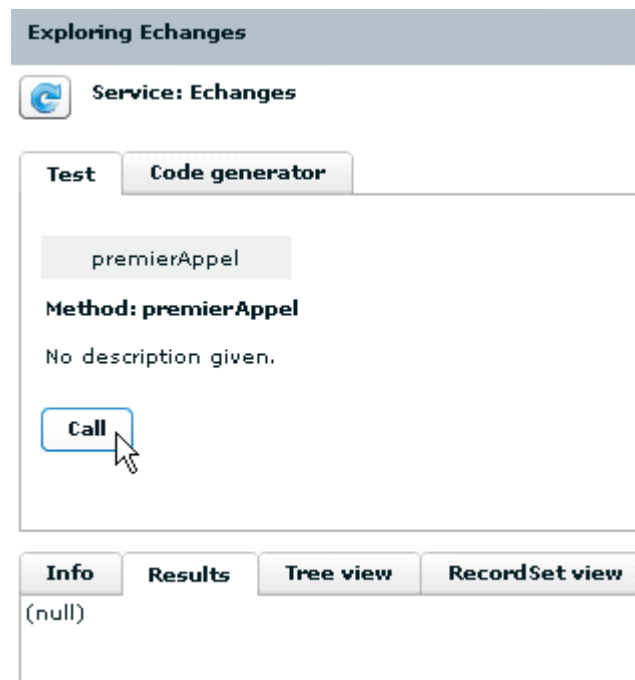


*Figure 19-4. Explorateur de service.*

En sélectionnant un service, la partie droite de l'application découvre les méthodes associées au service.

Chaque méthode est accessible et peut être testée directement depuis le navigateur. Cela permet de pouvoir développer la logique serveur sans avoir à tester depuis Flash.

L'onglet *Results* affiche alors le résultat de l'exécution de la méthode distante `premierAppel` :



*Figure 19-5. Explorateur de méthodes liées au service.*

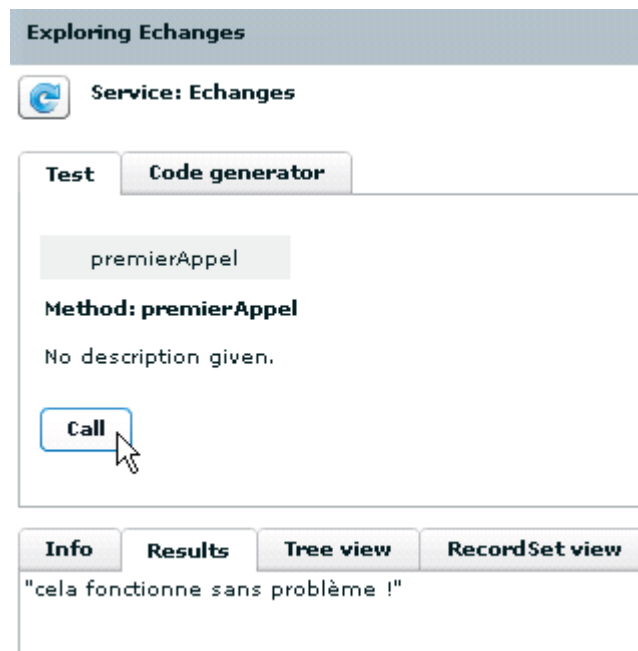
Dans notre exemple, la méthode `premierAppel` ne renvoie aucune valeur, l'onglet *Results* affiche donc `null`.

Afin de retourner des informations à Flash, la méthode doit *obligatoirement* utiliser le mot clé `return` :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }

    function premierAppel ()
    {
        return "cela fonctionne sans problème !";
    }
}
?>
```

Si nous testons à nouveau la méthode, nous pouvons voir directement les valeurs retournées :



*Figure 19-6. Explorateur de méthodes liées au service.*

La possibilité de pouvoir exécuter depuis le navigateur les méthodes distantes est un avantage majeur. L'explorateur service doit être considéré comme un véritable outil de débogage de service.

Au cas où une erreur PHP serait présente au sein du service, l'explorateur de service nous l'indique au sein de l'onglet *Results*

Dans le code suivant, nous oublions d'ajouter un point virgule en fin d'expression :

```
<?php

class Echanges
{

    function Echanges ( )

    {

    }

    function premierAppel ()

    {

        return "cela fonctionne sans problème !"

    }

}

?>
```

En tentant d'exécuter la méthode, le message suivant est affiché au sein de l'onglet *Results* :

```
Parse error: parse error, unexpected '}' in C:\Program Files\EasyPHP
2.0b1\www\echanges\services\org\bytearray\test\Echanges.php on line 18
```

Les onglets situés en dessous de l'onglet *Test* nous apportent d'autres informations comme les temps d'exécution ou le poids des données transférées.

La figure 19-7 illustre les informations liées à l'appel de la méthode *premierAppel* :

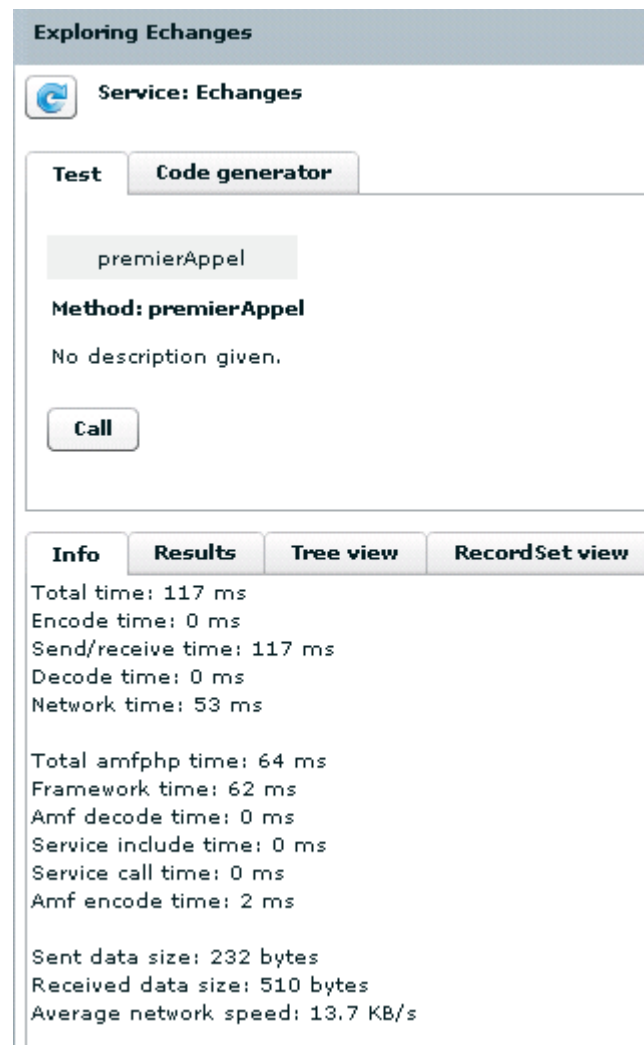


Figure 19-7. Onglet *Info*.

Imaginons que nous souhaitions afficher lors de l'exécution de la méthode `premierAppel` la longueur d'une chaîne.

Nous pouvons utiliser pour cela la méthode statique `trace` de la classe `NetDebug` intégrée à AMFPHP.

Dans le code suivant, nous affichons la longueur de la variable `$chaine` :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }
}
```

```
function premierAppel ()
{
    $chaine = "cela fonctionne sans problème !";
    NetDebug::trace( "infos persos : " . strlen ( $chaine ) );
}
}
?>
```

En exécutant la méthode, l'onglet *Trace* retourne le message personnalisé telle une fenêtre de sortie classique.

La figure 19-7 illustre le message affiché :



*Figure 19-7. Utilisation du débogage personnalisé.*

Cette fonctionnalité s'avère très précieuse lors de débogage de scripts, nous découvrirons l'intérêt des autres onglets très rapidement.

Si vous disposez de PHP 5 sur votre serveur, vous avez la possibilité d'utiliser les modificateurs de contrôle d'accès similaire à ActionScript 3.

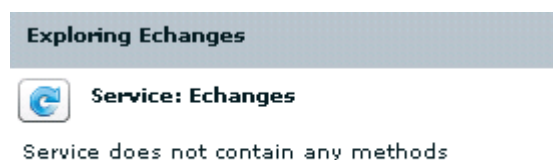
En définissant une méthode comme privée, celle-ci ne pourra plus être appelée depuis Flash, mais seulement depuis une autre méthode de la classe :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }

    private function premierAppel ( )
    {
        return "cela fonctionne sans problème !";
    }
}
?>
```

L'explorateur de méthodes considère alors que cette méthode n'est plus disponible depuis l'extérieur.

La figure 19-8 illustre l'état de l'explorateur de service :



*Figure 19-8. Aucune méthode disponible auprès du service.*

Nous retrouvons ici encore l'intérêt de la technologie Flash Remoting où nous évoluons dans un contexte orienté objet. L'utilisation de tels mécanismes améliore fortement la souplesse de développement d'une application dynamique.

Il est temps de coder quelques lignes d'ActionScript, nous allons dès maintenant nous connecter au service `Echanges` et exécuter la méthode `premierAppel`.

---

## A retenir

---

- Le service distant est une classe définissant différentes méthodes accessibles depuis Flash.
- L'explorateur de service nous permet de tester les méthodes en direct afin d'optimiser le débogage.

## Se connecter au service

Afin de se connecter à un service distant en ActionScript 3, nous disposons de différentes classes dont voici le détail :

- `flash.net.NetConnection` : la classe `NetConnection` permet de se connecter à un service distant
- `flash.net.Socket` : la classe `Socket` offre une connectivité TCP/IP offrant la possibilité d'échanger des données au format AMF à l'aide du protocole HTTP.

Afin d'appeler la méthode `premierAppel`, nous créons une instance de la classe `NetConnection` :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();
```

La classe `NetConnection` définit une propriété `objectEncoding` permettant de définir la version d'AMF utilisé pour les échanges.

En ActionScript 3, le format AMF3 est utilisé par défaut :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();

// affiche : 3
trace( connexion.objectEncoding );

// affiche : true
trace( connexion.objectEncoding == ObjectEncoding.AMF3 );
```

Au cas où nous souhaiterions nous connecter à une passerelle n'étant pas compatible avec AMF3, nous devons spécifier explicitement d'utiliser le format AMF0 :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();

// utilisation du format AMF0 pour les échanges
connexion.objectEncoding = ObjectEncoding.AMF0;
```

La version 1.9 d'AMFPHP étant compatible avec le format AMF3, nous ne modifions pas la propriété `objectEncoding` et nous connectons à la passerelle `gateway.php` à l'aide de la méthode `connect` :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();

// connexion à la passerelle AMFPHP
```

```
| connexion.connect ("http://localhost/echanges/gateway.php");
```

Même si l'adresse de la passerelle est erronée ou inaccessible, l'objet `NetConnection` ne renverra aucune erreur lors de l'appel de la méthode `connect`.

L'objet `NetConnection` diffuse différents événements dont voici le détail :

- `AsyncErrorEvent.ASYNC_ERROR` : diffusé lorsqu'une erreur asynchrone intervient.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque la transmission des données échoue.
- `NetStatusEvent.NET_STATUS` : diffusé lorsqu'une erreur fatale intervient.
- `SecurityErrorEvent.SECURITY_ERROR` : diffusé lorsque nous tentons d'appeler la méthode d'un service évoluant sur un autre domaine sans en avoir l'autorisation.

Nous écoutons les différents événements en passant une seule fonction écouteur afin de centraliser la gestion des erreurs :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();

// connexion à la passerelle amfphp
connexion.connect ("http://localhost/echanges/gateway.php");

// écoute des différents événements
connexion.addEventListener( NetStatusEvent.NET_STATUS, erreurConnexion);
connexion.addEventListener( IOErrorEvent.IO_ERROR, erreurConnexion);
connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion);
connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR, erreurConnexion);

function erreurConnexion ( pEvt:Event ):void
{
    trace( pEvt );
}
```

Nous devons maintenant gérer le retour du serveur, Flash CS3 intègre pour cela une classe spécifique.

## La classe `Responder`

Avant d'appeler une méthode distante, il convient de créer au préalable un objet de gestion de retour des appels grâce à la classe `flash.net.Responder`.

Le constructeur de la classe `Responder` possède la signature suivante :



```
public function Responder(result:Function, status:Function = null)
```

Seul le premier paramètre est obligatoire :

- **result** : référence à la fonction gérant le retour du serveur en cas de réussite de l'appel.
- **status** : référence à la fonction gérant le retour du serveur en cas d'échec de l'appel.

Dans le code suivant nous créons un objet **Responder** en passant deux fonctions **succes** et **echec** :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();

// connexion à la passerelle amfphp
connexion.connect ("http://localhost/echanges/gateway.php");

// écoute des différents événements
connexion.addEventListener( NetStatusEvent.NET_STATUS, erreurConnexion );
connexion.addEventListener( IOErrorEvent.IO_ERROR, erreurConnexion );
connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR, erreurConnexion );
connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR, erreurConnexion );

function erreurConnexion ( pEvt:Event ):void
{
    trace( pEvt );
}

// création des fonctions de gestion de retour serveur
function succes ( pRetour:* ):void
{
    trace("retour serveur");
}

function echec ( pErreur:* ):void
{
    trace("echec de l'appel");
}

// création d'un gestionnaire de retour des appels
var retourServeur:Responder = new Responder (succes, echec);
```

Une fois l'objet **Responder** définit, nous pouvons appeler la méthode distante.

---

## A retenir

---

- Les classes `NetConnection` et `Socket` permettent de se connecter à une passerelle remoting à l'aide de la méthode `connect`.
- En ActionScript 3, le format AMF3 est utilisé par défaut.
- Afin de gérer le retour du serveur, nous devons créer un objet `Responder`.
- Ce dernier nécessite deux fonctions qui seront déclenchées en cas de succès ou échec de l'appel.

## Appeler une méthode distante

Afin d'appeler la méthode distante, nous utilisons la méthode `call` de l'objet `NetConnection` dont voici la signature :

```
public function call(command:String, responder:Responder, ... arguments):void
```

Les deux premiers paramètres sont obligatoires :

- `command` : la méthode du service à appeler.
- `responder` : l'objet `Responder` traitant les retours serveurs.
- `arguments` : les paramètres à transmettre à la méthode distante.

Lorsque la méthode `call` est appelée, un paquet AMF est créé contenant le nom de la méthode à exécuter au sein du service ainsi que les données à transmettre.

Dans le code suivant, nous passons le chemin complet de la méthode à appeler ainsi qu'une instance de la classe `Responder` pour gérer le retour du serveur :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();

// connexion à la passerelle amfphp
connexion.connect ("http://localhost/echanges/gateway.php");

// création des fonctions de gestion de retour serveur
function succes ( pRetour:* ):void
{
    trace("retour serveur");
}

function echec ( pErreur:* ):void
{
    trace("echec de l'appel");
}

// création d'un objet de gestion de l'appel
var retourServeur:Responder = new Responder (succes, echec);
```

```
// appel de la méthode distante
connexion.call ("org.bytearray.test.Echanges.premierAppel", retourServeur);
```

En testant le code précédent, nous voyons que la fonction `succes` est exécutée.

Attention, n'oubliez pas que Flash fonctionne de manière asynchrone. La méthode `call` de l'objet `NetConnection` ne renvoie donc aucune valeur. Seul l'objet `Responder` passé en paramètre permet de gérer le résultat de l'appel.

Pour récupérer les données renvoyées par le serveur nous utilisons le paramètre `pRetour` de la fonction `succes` :

```
function succes ( pRetour:* ):void
{

    // affiche : cela fonctionne sans problème !
    trace ( pRetour );

}
```

Nous venons d'appeler notre première méthode distante par Flash Remoting en quelques lignes seulement.

Souvenez-vous, lors du chapitre 14 intitulé *Chargement et envoi de données* nous devions nous assurer manuellement du décodage et de l'encodage UTF-8 afin de préserver les caractères spéciaux.

Le format AMF encode les chaînes de caractères en UTF-8, et AMFPHP se charge automatiquement du décodage. Il n'est donc plus nécessaire de se soucier de l'encodage des caractères spéciaux avec Flash Remoting.

Nous allons à présent modifier la méthode `premierAppel` afin que celle-ci accepte un paramètre `$pMessage` :

```
<?php

class Echanges
{

    function Echanges ( )

    {

    }

    function premierAppel ( $pMessage )

    {

        return "vous avez dit : $pMessage ?";

    }

}
```

```

}
?>

```

Lorsqu'un paramètre est ajouté à une méthode, l'explorateur de service nous donne la possibilité de passer dynamiquement les valeurs au sein d'un champ texte de saisie :



*Figure 19-9. Paramètre.*

Si nous testons à nouveau le code ActionScript précédent, nous remarquons que la fonction `echec` est déclenchée.

La méthode `premierAppel` attend désormais un paramètre. En l'omettant, une erreur est levée que nous pouvons intercepter grâce à la fonction `echec` passée en paramètre à l'objet `Responder`.

En ciblant la propriété `description` de l'objet retourné nous obtenons des informations liées à l'erreur en cours :

```

function echec ( pErreur:* ):void
{
    // affiche : Missing argument 1 for Echanges::premierAppel()
    trace( pErreur.description ) ;
}

```

Les informations renvoyées par AMFPHP nous permettent ainsi de déboguer l'application plus facilement. Ce type de mécanisme illustre la puissance de Flash Remoting en matière de débogage.

Dans le cas présent, nous apprenons de manière explicite que nous avons omis le paramètre attendu de la méthode distante `premierAppel`.

Si nous itérons sur l'objet retourné nous découvrons de nouvelles propriétés :

```
function echec ( pErreur:* ):void
{
    /* affiche :
    details : C:\wamp\www\echanges\services\org\bytearray\test\Echanges.php
    level : Unknown error type
    description : Missing argument 1 for Echanges::premierAppel()
    line : 12
    code : AMFPHP_RUNTIME_ERROR
    */
    for ( var p:String in pErreur )

    {

        trace( p + " : " + pErreur[p] );

    }

}
```

Dans le code suivant, nous tentons d'appeler une méthode inexistante dû à une faute de frappe :

```
connexion.call ("org.bytearray.test.Echanges.premierApel", retourServeur);
```

La propriété `description` de l'objet retourné par le serveur nous indique qu'une telle méthode n'existe pas au sein du service :

```
function echec ( pErreur:* ):void
{

    // affiche : The method {premierApel} does not exist in class {Echanges}.
    trace( pErreur.description );

}
```

De la même manière, si nous nous trompons de service :

```
connexion.call ("org.bytearray.test.Echange.premierAppel", retourServeur);
```

AMFPHP nous oriente à nouveau vers la cause de l'erreur :

```
function echec ( pErreur:* ):void
{

    // affiche : The class {Echange} could not be found under the class path
    {C:\wamp\www\echanges\services\org\bytearray\test\Echange.php}
    trace( pErreur.description );

}
```

```
| }  
|
```

Nous allons à présent nous intéresser aux différents types de données échangeables et découvrir l'intérêt de la sérialisation automatique assurée par Flash Remoting.

## A retenir

- La méthode `call` de l'objet `NetConnection` permet d'appeler une méthode distante.
- Si l'appel réussit, la fonction `succes` passée en paramètre à l'objet `Responder` est exécutée.
- Dans le cas contraire, la fonction `echec` est déclenchée.
- Flash Remoting gère l'encodage de caractères spéciaux automatiquement.

## Echanger des données primitives

Comme nous l'avons expliqué précédemment, Flash Remoting se charge de la sérialisation et désérialisation des données.

Afin de nous rendre compte de ce comportement, nous modifions la méthode `premierAppel` en la renommant `echangesTypes` :

```
<?php  
class Echanges  
{  
    function Echanges ( )  
    {  
    }  
    function echangeTypes ( $pDonnees )  
    {  
        return gettype ( $pDonnees );  
    }  
}  
?>
```

Celle-ci accepte un paramètre et retourne désormais son type grâce à la méthode PHP `gettype`. Celle-ci est similaire à l'instruction `typeof` d'ActionScript 3.

Cette méthode va nous permettre de savoir sous quel type les données envoyées depuis Flash arrivent côté PHP.

Nous modifions les paramètres passés à la méthode `call` en spécifiant le nouveau nom de méthode ainsi que le valeur booléenne `true` :

```
connexion.call ("org.bytearray.test.Echanges.echangeTypes", retourServeur, true);
```

La fonction `succes` reçoit alors la chaîne `boolean` :

```
function succes ( pRetour:* ):void
{
    // affiche : boolean
    trace ( pRetour );
}
```

Nous voyons que la passerelle AMFPHP a automatiquement conservé le type booléen entre ActionScript et PHP, on dit alors que les données échangées sont *supportées*.

Mais que se passe t-il si AMFPHP ne trouve pas de type équivalent ?

Dans ce cas, AMFPHP interprète le type de données et convertit les données en un type équivalent.

Dans le code suivant, nous passons la valeur 5 de type `int` :

```
connexion.call ("org.bytearray.Echanges.echangeTypes", retourServeur, 5);
```

La méthode distante `echangeTypes` renvoie alors le type reçu, la fonction `succes` affiche les données retournées :

```
function succes ( pRetour:* ):void
{
    // affiche : double
    trace ( pRetour );
}
```

Nous voyons que la passerelle AMFPHP a automatiquement convertit le type `int` en `double`, c'est-à-dire son équivalent PHP.

La valeur 5 est envoyée au sein d'un paquet AMF à la méthode distante, AMFPHP déséréalise automatiquement le paquet AMF et convertit le type `int` en un type compatible PHP. On dit alors que les données échangées sont *interprétées*.

Le tableau ci-dessous regroupe les différents types primitifs supportés et interprétés par AMFPHP :

Type ActionScript	Type PHP	Conversion automatique	Version AMF

null	NULL	Oui	AMF0 / AMF3
int	double	Oui	AMF3
uint	double	Oui	AMF3
Number	double	Oui	AMF0 / AMF3
Boolean	boolean	Oui	AMF0 / AMF3
String	string	Oui	AMF0 / AMF3

*Tableau 2. Tableau des types primitifs supportés et interprétés.*

Cette conversion automatique permet d'échanger des données primitives sans aucun travail de notre part. AMFPHP se charge de toute la sérialisation dans les deux sens, ce qui nous permet de nous concentrer sur d'autres parties plus importantes de l'application telles que l'interface ou autres.

Sans Flash Remoting, nous aurions du sérialiser les données sous la forme d'une chaîne de caractères, puis désérialiser les données côté serveur manuellement.

Grâce à AMFPHP, ce processus est simplement supprimé.

Bien entendu, dans la plupart des applications dynamiques, nous n'échangeons pas de simples données comme celle-ci.

Si nous souhaitons concilier l'avantage des deux technologies, nous pouvons faire transiter une chaîne de caractères compressée au sein d'AMF puis transformer celle-ci en objet XML au sein de Flash.

Nous ajoutons une nouvelle méthode `recupereMenu` :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }

    function exchangeTypes ( $pDonnees )
    {
```



```
        return gettype ( $pDonnees );
    }

    function recupereMenu ()
    {
        $menuXML = '<MENU>
            <RUBRIQUE titre = "Nouveautés" id="12">
                <RUBRIQUE titre = "CD" id="487"/>
                <RUBRIQUE titre = "Vinyls" id="540"/>
            </RUBRIQUE>
            <RUBRIQUE titre = "Concerts" id="25">
                <RUBRIQUE titre = "Funk" id="15"/>
                <RUBRIQUE titre = "Soul" id="58"/>
            </RUBRIQUE>
        </MENU>';

        return $menuXML;
    }
}

?>
```

En appelant la méthode `recupereMenu`, nous recevons la chaîne de caractère que nous transformons aussitôt en objet XML :

```
// création de la connexion
var connexion:NetConnection = new NetConnection ();

// connexion à la passerelle amfphp
connexion.connect ("http://localhost/echanges/gateway.php");

// création des fonctions de gestion de retour serveur
function succes ( pRetour:* ):void
{
    try
    {
        // conversion de la chaîne en objet XML
        var menu:XML = new XML ( pRetour );

        /* affiche :
        <RUBRIQUE titre="Concerts" id="25">
            <RUBRIQUE titre="Funk" id="15"/>
            <RUBRIQUE titre="Soul" id="58"/>
        </RUBRIQUE>
        */
        trace(menu.RUBRIQUE.(@id == 25) );

        //affiche : Funk
        trace(menu..RUBRIQUE.(@id == 15).@titre );

    } catch ( pErreur:Error )
    {
    }
}
```

```
        trace( "erreur d'interprétation du flux XML");
    }
}

function echec ( pErreur:* ):void
{
    trace("echec de l'appel");
}

// création d'un objet de gestion de l'appel
var retourServeur:Responder = new Responder (succes, echec);

// appel de la méthode recupereMenu distante, récupération du flux xml du menu
connexion.call ("org.bytearray.test.Echanges.recupereMenu", retourServeur);
```

En utilisant cette approche, nous conservons :

- La puissance de débogage de Flash Remoting.
- L'appel de méthodes distantes directement depuis Flash.
- La simplicité du format XML.
- La puissance de la norme E4X.

Même si la conversion de la chaîne en objet XML par ActionScript pourrait ralentir les performances de l'application si le flux XML devient important, cela resterait minime dans notre exemple.

Il peut être nécessaire d'échanger des données plus complexes, nous allons voir que Flash Remoting va à nouveau nous faciliter les échanges.

## A retenir

- Grâce à Flash Remoting les données primitives sont automatiquement sérialisées et désérialisées.
- Lorsqu'AMFPHP trouve un type correspondant, on dit que les données sont supportées.
- Dans le cas contraire, AMFPHP interprète les données en un type équivalent.
- Il est tout à fait possible de concilier un format de représentation de données tel XML avec Flash Remoting.
- Le compromis XML et AMF est une option élégante à prendre en considération.

### Echanger des données composites

Dans la plupart des applications dynamiques, nous souhaitons généralement échanger des données plus complexes, comme des tableaux ou des objets associatifs.

Voici le tableau des différents types composites supportés et interprétés par AMFPHP :

Type ActionScript	Type PHP	Conversion automatique	Version AMF
Object	associative array	Oui	AMF0 / AMF3
Array	array	Oui	AMF0 / AMF3
XML (E4X)	string	Non	AMF3
XMLDocument	string	Non	AMF0 / AMF3
Date	double	Non	AMF0 / AMF3
RecordSet	Ressource MySQL	Oui (Uniquement de MySQL vers Flash)	AMF0 / AMF3
ByteArray	ByteArray (classe AMFPHP)	Oui	AMF3

*Tableau 3. Tableau des types composites supportés et interprétés.*

Dans le code suivant nous envoyons à la méthode distante un objet

`Date` :

```
connexion.call ("org.bytearray.test.Echanges.echangeTypes", retourServeur, new Date());
```

L'objet `Date` est transmis, AMFPHP ne trouvant pas de type équivalent convertit alors l'objet `Date` en timestamp Unix compatible.

Ainsi, nous recevons côté PHP un `double` :

```
function succes ( pRetour:* ):void
{
    // affiche : double
    trace( pRetour );
}
```

Très souvent, nous avons besoin d'envoyer au serveur un objet contenant différentes informations.

Dans le code suivant, nous envoyons au serveur un tableau associatif contenant les informations liées à un utilisateur :

```
// création d'un joueur fictif
var joueur:Object = new Object();

joueur.nom = "Groove";
joueur.prenom = "Bob";
joueur.age = 29;
joueur.ville = "Paris";
joueur.points = 35485;

// appel de la méthode distante, le joueur est envoyé à la méthode distante
connexion.call ("org.bytearray.test.Echanges.echangeTypes", retourServeur, joueur);
```

Nous modifions la méthode distante en renvoyant simplement les données passées en paramètre :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }

    function echangeTypes ( $pDonnees )
    {
```

```
        return $pDonnees;
    }
}
?>
```

L'objet est retourné à Flash sous sa forme originale :

```
function succes ( pRetour:* ):void
{
    /* affiche :
    nom   : Groove
    prenom : Bob
    age   : 29
    ville : Paris
    points : 35485
    */
    for ( var p:String in pRetour )
    {
        trace( p, " : " + pRetour[p] );
    }
}
```

Nous remarquons donc que la sérialisation automatique est opérée dans les deux sens et nous permet de gagner un temps précieux.

Coté PHP nous accédons aux propriétés de l'objet de manière traditionnelle. Nous pouvons par exemple augmenter l'âge et le score du joueur passé en paramètre :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }

    function exchangeTypes ( $pDonnees )
    {
        $pDonnees["age"] += 10;
        $pDonnees["points"] += 500;

        return $pDonnees;
    }
}
?>
```

L'objet est retourné modifié au lecteur Flash :

```
function succes ( pRetour:* ):void
{
    /* affiche :
    nom : Groove
    ville : Paris
    prenom : Bob
    age : 39
    points : 35985
    */
    for ( var p:String in pRetour )
    {
        trace( p, " : " + pRetour[p] );
    }
}
```

Grâce au format AMF3 et la version 1.9 d'AMFPHP, il est possible d'échanger des objets de types `ByteArray`.

Dans le code suivant, nous créons un tableau d'octets vierge, puis nous écrivons des données texte :

```
// création d'un tableau d'octets vierge
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Voilà du texte encodé en binaire !");
```

L'instance de `ByteArray` est ensuite transmise à la méthode distante :

```
// création d'un tableau d'octets vierge
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Voilà du texte encodé en binaire !");

// appel de la méthode distante, le ByteArray est envoyé au serveur
connexion.call ("org.bytearray.test.Echanges.echangeTypes", retourServeur,
fluxBinaire);
```

Lorsque la fonction `succes` est exécutée, l'objet `ByteArray` retourné par le serveur est intact et les données préservées :

```
function succes ( pRetour:* ):void
{
    // affiche : true
    trace( pRetour is ByteArray );

    // affiche : Voilà du texte encodé en binaire !
    trace( pRetour.readUTFBytes ( pRetour.bytesAvailable ) );
}
```

Bien que l'intérêt puisse paraître limité, cette fonctionnalité s'avère extrêmement puissante. Nous reviendrons très bientôt sur l'intérêt d'échanger des instances de `ByteArray`.

Malheureusement, certains objets ne peuvent être sérialisés au format AMF, c'est le cas des objets de type `DisplayObject`.

En modifiant la méthode `exchangeTypes` nous pourrions penser pouvoir renvoyer l'objet graphique transmis :

```
<?php
class Echanges
{
    function Echanges ( )
    {
    }

    function exchangeTypes ( $pDonnees )
    {
        return $pDonnees;
    }
}
?>
```

Si nous tentons de passer une instance de `MovieClip` à cette même méthode :

```
// création d'une instance de MovieClip
var animation:MovieClip = new MovieClip();

// appel de la méthode distante, un objet graphique est passé à la méthode distante
connexion.call ("org.bytearray.test.Echanges.exchangeTypes", retourServeur,
animation);
```

La sérialisation AMF échoue, la méthode distante `exchangeTypes` renvoie la valeur `null` :

```
function succes ( pRetour:* ):void
{
    // affiche : null
    trace( pRetour );

    // affiche : true
    trace( pRetour == null )
}
```

Cette limitation n'est pas due à la passerelle remoting AMFPHP mais au format AMF qui ne gère pas au jour d'aujourd'hui la sérialisation et désérialisation d'objets graphiques.

Dans certains cas, nous pouvons avoir besoin de transmettre un flux XML. En passant un objet de type XML, celui-ci est alors aplati sous la forme d'une chaîne de caractères UTF-8 :

```
// création d'un objet XML
var donneesXML:XML = <SCORE><JOUEUR id='25' score='15888'/></SCORE>;

// appel de la méthode echangeTypes distante, nous transmettons un objet XML
connexion.call ("org.bytearray.test.Echanges.echangeTypes", retourServeur,
donneesXML);
```

Le flux XML revient sous la forme d'une chaîne de caractères :

```
function succes ( pRetour:* ):void
{
    // l'objet XML a été converti sous forme de chaîne de caractères
    trace( pRetour is String );
}
```

AMFPHP préfère conserver une chaîne et nous laisser gérer la manipulation de l'arbre XML.

## A retenir

- Grâce à Flash Remoting les données composites sont automatiquement sérialisées et désérialisées.
- Lorsqu'AMFPHP trouve un type correspondant, on dit que les données sont supportées.
- Dans le cas contraire, AMFPHP interprète les données en un type équivalent.
- Les objets graphiques ne peuvent ne sont pas compatibles avec le format AMF.

## Echanger des données typées

Au cas où nous souhaiterions échanger des types personnalisés avec la passerelle Remoting, AMFPHP intègre un mécanisme approprié.

Une instance de classe personnalisée de type `Utilisateur` peut par exemple être définie côté Flash et transmise au service distant, en conservant côté PHP le type `Utilisateur`.

De la même manière, une méthode distante peut renvoyer à Flash des instances de classes en assurant une conservation des types personnalisés.



Pour plus d'informations, rendez vous à l'adresse suivante :

<http://amfphp.org/docs/classmapping.html>

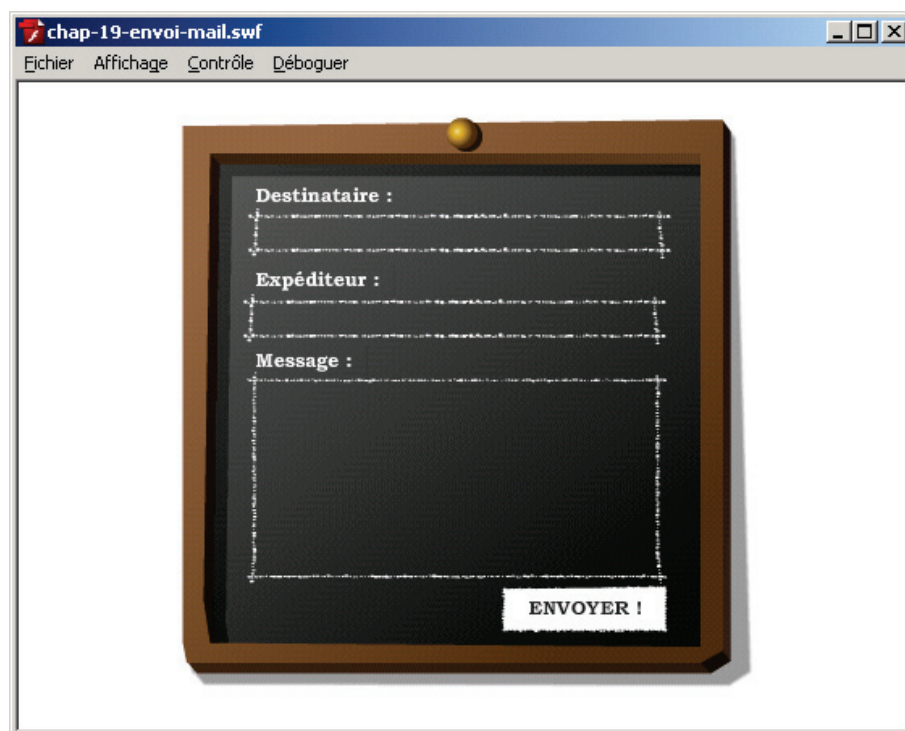
Passons maintenant de la théorie à la pratique.

## Envoyer un email avec Flash Remoting

Au cours du chapitre 14 intitulé *Chargement et envoi de données* nous avons développé un formulaire permettant d'envoyer un email depuis Flash.

Nous allons reprendre l'application et remplacer les échanges réalisés à l'aide la classe `URLLoader` par un échange par Flash Remoting.

La figure 19-10 illustre le formulaire :



*Figure 19-10. Formulaire d'envoi d'email.*

La classe de document suivante est associée :

```
package org.bytearray.document

{

    import flash.text.TextField;
    import flash.display.SimpleButton;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault
```

```
{  
  
    public var destinataire:TextField;  
    public var sujet:TextField;  
    public var message:TextField;  
    public var boutonEnvoi:SimpleButton;  
  
    public function Document ()  
    {  
  
    }  
  
}
```

Afin de pouvoir envoyer notre email, nous devons tout d'abord prévoir une méthode d'envoi. Pour cela nous ajoutons une méthode `envoiMessage` à notre service distant :

```
<?php  
class Echanges {  
  
    function exchangeTypes ( $pDonnees )  
    {  
  
        return $pDonnees;  
    }  
  
    function envoiMessage ( $pInfos )  
    {  
  
        $destinaire = $pInfos["destinataire"];  
        $sujet = $pInfos["sujet"];  
        $message = $pInfos["message"];  
  
        return @mail ( $destinaire, $sujet, $message );  
    }  
}  
?>
```

La méthode `envoiMessage` reçoit un tableau associatif contenant les informations nécessaires et procède à l'envoi du message.

La valeur renvoyée par la fonction PHP `mail` indiquant le succès ou l'échec de l'envoi est retournée à Flash.

Afin d'appeler la méthode `envoiMessage` nous ajoutons les lignes suivantes à la classe de document définie précédemment :

```
| package org.bytearray.document
```

---

```
{

    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.SecurityErrorEvent;
    import flash.events.AsyncErrorEvent;
    import flash.events.NetStatusEvent;
    import flash.net.NetConnection;
    import flash.text.TextField;
    import flash.display.SimpleButton;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault

    {

        public var destinataire:TextField;
        public var sujet:TextField;
        public var message:TextField;
        public var boutonEnvoi:SimpleButton;
        private var connexion:NetConnection;

        public function Document ()

        {

            //création de la connexion
            connexion = new NetConnection();

            // écoute des différents événements
            connexion.addEventListener( NetStatusEvent.NET_STATUS,
erreurConnexion );
            connexion.addEventListener( IOErrorEvent.IO_ERROR,
erreurConnexion );
            connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion );
            connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
erreurConnexion );

        }

        private function erreurConnexion ( pEvt:Event ):void

        {

            trace( pEvt );

        }

    }

}
```

Puis nous ajoutons une propriété constante **PASSERELLE** contenant l'adresse de la passerelle et nous nous connectons à celle-ci :

```
package org.bytearray.document

{

    import flash.events.Event;
```

```
import flash.events.IOErrorEvent;
import flash.events.SecurityErrorEvent;
import flash.events.AsyncErrorEvent;
import flash.events.NetStatusEvent;
import flash.text.TextField;
import flash.display.SimpleButton;
import flash.net.NetConnection;
import org.bytearray.abstrait.ApplicationDefault;

public class Document extends ApplicationDefault
{
    public var destinataire:TextField;
    public var sujet:TextField;
    public var message:TextField;
    public var boutonEnvoi:SimpleButton;
    private var connexion:NetConnection;

    // adresse de la passerelle AMFPHP
    private static const PASSERELLE:String =
"http://localhost/echanges/gateway.php";

    public function Document ()
    {
        //création de la connexion
        connexion = new NetConnection();

        // écoute des différents événements
        connexion.addEventListener( NetStatusEvent.NET_STATUS,
erreurConnexion );
        connexion.addEventListener( IOErrorEvent.IO_ERROR,
erreurConnexion );
        connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion );
        connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
erreurConnexion );

        // connexion à la passerelle
        connexion.connect ( Document.PASSERELLE );
    }

    private function erreurConnexion ( pEvt:Event ):void
    {
        trace( pEvt );
    }
}
```

Nous ajoutons une instance de **Responder** afin de gérer les retours serveurs :

```
package org.bytearray.document
```

```
{

    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.SecurityErrorEvent;
    import flash.events.AsyncErrorEvent;
    import flash.events.NetStatusEvent;
    import flash.net.Responder;
    import flash.text.TextField;
    import flash.display.SimpleButton;
    import flash.net.NetConnection;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault

    {

        public var destinataire:TextField;
        public var sujet:TextField;
        public var message:TextField;
        public var boutonEnvoi:SimpleButton;
        private var connexion:NetConnection;
        private var retourServeur:Responder;

        // adresse de la passerelle AMFPHP
        private static const PASSERELLE:String =
"http://localhost/echanges/gateway.php";

        public function Document ()

        {

            //création de la connexion
            connexion = new NetConnection();

            // création de l'objet de gestion des retours serveur
retourServeur = new Responder ( succes, echec );

            // écoute des différents événements
            connexion.addEventListener( NetStatusEvent.NET_STATUS,
ecouteurCentralise );
            connexion.addEventListener( IOErrorEvent.IO_ERROR,
ecouteurCentralise );
            connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
ecouteurCentralise );
            connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
ecouteurCentralise );

            // connexion à la passerelle
            connexion.connect ( Document.PASSERELLE );

        }

        private function succes ( pRetour:* ):void

        {

            trace ( pRetour );

        }

        private function echec ( pErreur:* ):void

    }

}
```

```
        {  
            trace ( pErreur );  
        }  
        private function ecouteurCentralise ( pEvt:Event ):void  
        {  
            trace( pEvt );  
        }  
    }  
}
```

Puis nous appelons la méthode distante `envoiMessage` lorsque le bouton `boutonEnvoi` est cliqué :

```
package org.bytearray.document  
{  
    import flash.events.Event;  
    import flash.events.MouseEvent;  
    import flash.events.IOErrorEvent;  
    import flash.events.SecurityErrorEvent;  
    import flash.events.AsyncErrorEvent;  
    import flash.events.NetStatusEvent;  
    import flash.net.Responder;  
    import flash.text.TextField;  
    import flash.display.SimpleButton;  
    import flash.net.NetConnection;  
    import org.bytearray.abstrait.ApplicationDefault;  
  
    public class Document extends ApplicationDefault  
    {  
        public var destinataire:TextField;  
        public var sujet:TextField;  
        public var message:TextField;  
        public var boutonEnvoi:SimpleButton;  
        private var connexion:NetConnection;  
        private var retourServeur:Responder;  
        private var infos:Object;  
  
        // adresse de la passerelle AMFPHP  
        private static const PASSERELLE:String =  
        "http://localhost/echanges/gateway.php";  
  
        public function Document ()  
        {  
            //création de la connexion  
            connexion = new NetConnection();  
  
            // création de l'objet de gestion des retours serveur
```

```
        retourServeur = new Responder ( succes, echec );

        // écoute des différents événements
        connexion.addEventListener( NetStatusEvent.NET_STATUS,
erreurConnexion );
        connexion.addEventListener( IOErrorEvent.IO_ERROR,
erreurConnexion );
        connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion );
        connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
erreurConnexion );

        // connexion à la passerelle
        connexion.connect ( Document.PASSERELLE );

        boutonEnvoi.addEventListener ( MouseEvent.CLICK, envoiMail );
    }

    private function erreurConnexion ( pEvt:Event ):void
    {
        trace( pEvt );
    }

    private function succes ( pRetour:* ):void
    {
        trace ( pRetour );
    }

    private function echec ( pErreur:* ):void
    {
        trace ( pErreur );
    }

    private function envoiMail ( pEvt:MouseEvent ):void
    {
        infos = new Object();

        infos.destinataire = destinataire.text;
        infos.sujet = sujet.text;
        infos.message = message.text;

        connexion.call ( "org.bytearray.Echanges.envoiMessage",
retourServeur, infos );
    }
}

}
```

En testant le code précédent, la méthode écouteur `succes` est déclenchée et reçoit la valeur `true` du serveur indiquant que le mail a bien été envoyé.

Afin de finaliser cet exemple, nous pouvons utiliser la classe `OutilsFormulaire` développée au cours du chapitre 14 et l'importer :

```
| import org.bytearray.ouutils.FormulaireOutils;
```

Puis nous modifions la méthode `envoiMail` afin de tester la validité de l'adresse saisie :

```
| private function envoiMail ( pEvt:MouseEvent ):void
| {
|     if ( FormulaireOutils.verifieEmail ( destinataire.text ) )
|     {
|         infos = new Object();
|         infos.destinataire = destinataire.text;
|         infos.sujet = sujet.text;
|         infos.message = message.text;
|
|         connexion.call ( "org.bytearray.Echanges.envoiMessage", retourServeur,
| infos );
|
|         } else destinataire.text = "Email non valide !";
|     }
| }
```

Afin de valider l'envoi du message au sein de l'application nous ajoutons la condition suivante au sein de la méthode de retour `succes` :

```
| private function succes ( pRetour:* ):void
| {
|     if ( pRetour ) message.text = "Message bien envoyé !";
|
|     else message.text = "Erreur d'envoi du message";
| }
| }
```

Ainsi, nous travaillons de manière transparente avec le script serveur sans avoir à nous soucier de sérialiser et désérialiser les données envoyées et reçues.

## Exporter une image

Nous avons vu précédemment qu'il était possible de transmettre une instance de `ByteArray` par Flash Remoting.



Comme nous le verrons au cours du chapitre 21 intitulé *ByteArray* il est possible de générer en ActionScript 3 un flux binaire grâce à la classe *ByteArray*.

Afin de pouvoir exploiter ce flux binaire, nous pouvons le transmettre au service distant afin que celui-ci le sauvegarde sur le serveur.

Nous allons reprendre l'application de dessin développée au cours du chapitre 9 intitulé *Etendre les classes natives* et ajouter une fonctionnalité d'export de notre dessin sous la forme d'une image PNG.

Nous définissons une classe de document associée :

```
package org.bytearray.document
{
    import flash.display.SimpleButton;
    import flash.display.Sprite;
    import org.bytearray.abstrait.ApplicationDefaut;

    public class Document extends ApplicationDefaut
    {
        private var dessin:Sprite;
        private var stylo:Stylo;
        public var boutonEnvoi:SimpleButton;

        public function Document ()
        {
            // création du conteneur de tracés vectoriels
            dessin = new Sprite();

            // ajout du conteneur à la liste d'affichage
            addChild ( dessin );

            // création du symbole
            stylo = new Stylo( .1 );

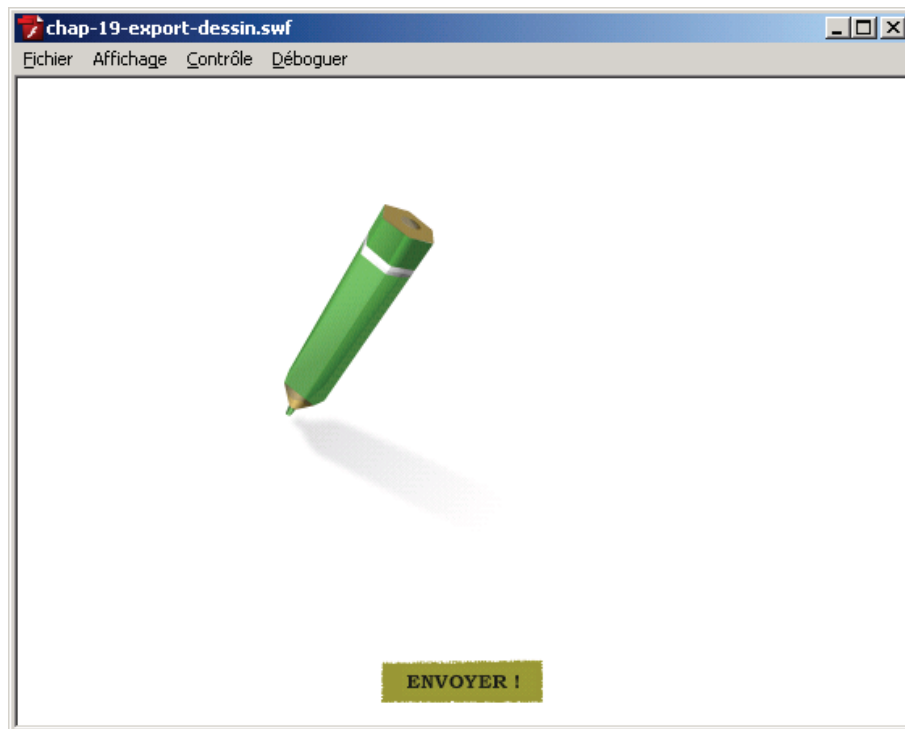
            // passage du conteneur de tracés
            stylo.affecteToile ( dessin );

            // ajout du symbole à la liste d'affichage
            addChild ( stylo );

            // positionnement en x et y
            stylo.x = 250;
            stylo.y = 200;
        }
    }
}
```

Nous ajoutons un bouton `boutonEnvoi`, permettant d'exporter le dessin sous forme bitmap.

La figure 19-11 illustre l'application :



*Figure 19-11. Application de dessin.*

Lors du clic bouton nous devons générer une image bitmap du dessin vectoriel, pour cela nous allons utiliser une classe d'encodage d'image `EncodeurPNG`.

Nous écoutons l'événement `MouseEvent.CLICK` du bouton d'export :

```
package org.bytearray.document
{
    import flash.display.SimpleButton;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault
    {
        private var dessin:Sprite;
        private var stylo:Stylo;
        private var renduBitmap:BitmapData;

        public function Document ()
```

```
{

    // création du conteneur de tracés vectoriels
    dessin = new Sprite();

    // ajout du conteneur à la liste d'affichage
    addChild ( dessin );

    // création du symbole
    stylo = new Stylo( .1 );

    // passage du conteneur de tracés
    stylo.affecteToile ( dessin );

    // ajout du symbole à la liste d'affichage
    addChild ( stylo );

    // positionnement en x et y
    stylo.x = 250;
    stylo.y = 200;

    boutonEnvoi.addEventListener ( MouseEvent.CLICK, exportBitmap );

}

private function exportBitmap ( pEvt:MouseEvent ):void
{

    trace("export bitmap");

}

}
```

Afin d'encoder notre dessin vectoriel en image PNG nous devons tout d'abord rendre sous forme bitmap les tracés vectoriels.

Souvenez-vous, nous avons découvert lors du chapitre 12 intitulé *Programmation bitmap* qu'il était possible de rasteriser un élément vectoriel grâce à la méthode `draw` de la classe `BitmapData`.

Nous modifions la classe de document afin d'intégrer une rasterisation du dessin vectoriel lorsque le bouton d'export est cliqué :

```
package org.bytearray.document

{

    import flash.display.BitmapData;
    import flash.display.SimpleButton;
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    import flash.utils.ByteArray;
    import org.bytearray.abstrait.ApplicationDefaut;
    import org.bytearray.encodage.images.EncodeurPNG;

    public class Document extends ApplicationDefaut
```

```
{

    private var dessin:Sprite;
    private var stylo:Stylo;
    private var renduBitmap:BitmapData;
    public var boutonEnvoi:SimpleButton;

    public function Document ()
    {

        // création du conteneur de tracés vectoriels
        dessin = new Sprite();

        // ajout du conteneur à la liste d'affichage
        addChild ( dessin );

        // création du symbole
        stylo = new Stylo( .1 );

        // passage du conteneur de tracés
        stylo.affecteToile ( dessin );

        // ajout du symbole à la liste d'affichage
        //addChild ( stylo );

        // positionnement en x et y
        stylo.x = 250;
        stylo.y = 200;

        boutonEnvoi.addEventListener ( MouseEvent.CLICK, exportBitmap );

    }

    private function exportBitmap ( pEvt:MouseEvent ):void
    {

        // création d'une image bitmap vierge
        renduBitmap = new BitmapData ( stage.stageWidth,
stage.stageHeight );

        // rasterisation des tracés
        renduBitmap.draw ( dessin );

        // encodage de l'image bitmap au format PNG
        var fluxBinaire:ByteArray = EncodeurPNG.encode ( renduBitmap );

        // affiche : 1488
        trace( fluxBinaire.length );

    }

}
```

La variable `fluxBinaire` représente un objet `ByteArray` contenant l'image encodée au format PNG, nous devons à présent transmettre le flux de l'image.

Nous ajoutons une nouvelle méthode `sauveImage` à notre service distant :

```
<?php

class Echanges
{

    function Echanges ( )

    {

    }

    function exchangeTypes ( $pDonnees )

    {

        return $pDonnees;

    }

    function sauveImage ( $pFluxImage )

    {

        //le ByteArray est placé au sein de la propriété data de l'objet reçu
        $flux = $pFluxImage->data;

        // nous décompressons le flux compressé coté Flash
        $flux = gzuncompress($flux);

        $nomImage = "capture.png";

        // sauvegarde de l'image
        $fp = @fopen("../../../images/".$nomImage, 'wb');
        $ecriture = @fwrite($fp, $flux);
        @fclose($fp);

        return $ecriture !== FALSE;

    }

}

?>
```

La méthode `sauveImage` reçoit le flux binaire en paramètre sous ma forme d'une instance de la classe `ByteArray` intégrée à AMFPHP.

Le flux est accessible par la propriété `data` de l'instance de `ByteArray`, nous sauvegardons le flux à l'aide des fonctions d'écriture PHP `fopen` et `fwrite`.

Attention, veuillez à créer un répertoire nommé `images`, afin d'accueillir les futures images sauvées. Dans notre exemple, ce dernier est placé au même niveau que le répertoire `services`.

Nous modifions la méthode `exportBitmap` afin de transmettre l'image à la méthode `sauveImage` :

```
package org.bytearray.document
```

---

```
{

import flash.display.BitmapData;
import flash.display.SimpleButton;
import flash.display.Sprite;
import flash.events.Event;
import flash.events.MouseEvent;
import flash.events.IOErrorEvent;
import flash.events.SecurityErrorEvent;
import flash.events.AsyncErrorEvent;
import flash.events.NetStatusEvent;
import flash.utils.ByteArray;
import flash.net.Responder;
import flash.net.NetConnection;
import org.bytearray.abstrait.ApplicationDefault;
import org.bytearray.encodage.images.EncodeurPNG;

public class Document extends ApplicationDefault

{

    private var dessin:Sprite;
    private var stylo:Stylo;
    private var renduBitmap:BitmapData;
    public var boutonEnvoi:SimpleButton;

    private var connexion:NetConnection;
    private var retourServeur:Responder;

    // adresse de la passerelle AMFPHP
    private static const PASSERELLE:String =
"http://localhost/echanges/gateway.php";

    public function Document ()
    {

        //création de la connexion
        connexion = new NetConnection();

        // création de l'objet de gestion des retours serveur
        retourServeur = new Responder ( succes, echec );

        // écoute des différents événements
        connexion.addEventListener( NetStatusEvent.NET_STATUS,
erreurConnexion );
        connexion.addEventListener( IOErrorEvent.IO_ERROR,
erreurConnexion );
        connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion );
        connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
erreurConnexion );

        // connexion à la passerelle
        connexion.connect ( Document.PASSERELLE );

        // création du conteneur de tracés vectoriels
        dessin = new Sprite();

        // ajout du conteneur à la liste d'affichage
        addChild ( dessin );
    }
}
```

```
// création du symbole
stylo = new Stylo( .1 );

// passage du conteneur de tracés
stylo.affecteToile ( dessin );

// ajout du symbole à la liste d'affichage
addChild ( stylo );

// positionnement en x et y
stylo.x = 250;
stylo.y = 200;

boutonEnvoi.addEventListener ( MouseEvent.CLICK, exportBitmap );

}

private function erreurConnexion ( pEvt:Event ):void
{
    trace( pEvt );
}

private function succes ( pRetour:* ):void
{
    if ( pRetour ) trace("image sauvegardée !");
    else trace("erreur d'enregistrement");
}

private function echec ( pErreur:* ):void
{
    trace( pErreur );
}

private function exportBitmap ( pEvt:MouseEvent ):void
{
    // création d'une image bitmap vierge
    renduBitmap = new BitmapData ( stage.stageWidth,
    stage.stageHeight );

    // rasterisation des tracés
    renduBitmap.draw ( dessin );

    // encodage de l'image bitmap au format PNG
    var fluxBinaire:ByteArray = EncodeurPNG.encode ( renduBitmap );

    // compression zlib du flux
    fluxBinaire.compress();

    // transmission du ByteArray par Flash Remoting
    connexion.call ( "org.bytearray.test.Echanges.sauveImage",
    retourServeur, fluxBinaire );
```

```

    }
}

```

Nous dessinons quelques tracés, puis nous cliquons sur le bouton d'export comme l'illustre la figure 19-12 :




Figure 19-12. Dessin.

Voici les différentes étapes lorsque nous cliquons sur le bouton d'export :

- Les tracés vectoriels sont rasterisés sous la forme d'un objet `BitmapData`.
- Une image PNG est générée à l'aide des pixels accessibles depuis l'objet `BitmapData`.
- Le flux de l'image est envoyé au service distant qui se charge de la sauvegarder sur le serveur.

Lorsque la méthode `succes` est déclenchée, l'image est sauvegardée sur le serveur. En accédant au répertoire `images` nous découvrons l'image sauvegardée comme l'illustre la figure 19-13 :

Nom	Taille	Type
 capture.jpg	4 Ko	Image JPEG



*Figure 19-13. Image sauvegardée.*

L'image pourrait aussi être sauvee directement en base de données au sein d'un champ de type `BLOB`. L'objet `ByteArray` pourrait ainsi être récupéré plus tard et affiché grâce à la méthode `loadBytes` de l'objet `Loader`.

Il serait aussi envisageable de transmettre uniquement les pixels de l'image à l'aide de la méthode `getPixels` de la classe `BitmapData`. Puis de générer n'importe quel type d'image côté serveur à l'aide d'une librairie spécifique telle GD ou autres.

## Se connecter à une base de données

Flash Remoting prend tout son sens lors de la manipulation d'une base de données.

Nous allons intégrer Flash Remoting au menu développé au cours du chapitre 7 intitulé *Interactivité*.

Pour cela nous devons définir une base de données, nous utiliserons dans cet exemple une base de données MySQL.

*Figure 19-14. Base de donnée MySQL.*

Nous créons une base de données intitulée `maBase` puis nous créons une table associée `menu`.

Afin de créer celle-ci nous pouvons exécuter la requête MySQL suivante :

```
--  
-- Structure de la table `menu`  
--  
  
CREATE TABLE `menu` (  
  `id` int(11) NOT NULL auto_increment,  
  `intitule` varchar(30) NOT NULL default '',  
  `couleur` int(11) NOT NULL default '0',
```

```











PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 AUTO_INCREMENT=6 ;

--
-- Contenu de la table `menu`
--

INSERT INTO `menu` (`id`, `intitule`, `couleur`) VALUES (1, 'Accueil',
16737536),
(2, 'Nouveautés', 16737536),
(3, 'Photos', 16737536),
(4, 'Liens', 16737536),
(5, 'Contact', 16737536);

```

Quelques données sont présentes dans la table `menu` :

←T→			id	intitule	couleur
<input type="checkbox"/>			1	Accueil	16737536
<input type="checkbox"/>			2	Nouveautés	16737536
<input type="checkbox"/>			3	Photos	16737536
<input type="checkbox"/>			4	Liens	16737536
<input type="checkbox"/>			5	Contact	16737536

*Figure 19-15. Données de la table menu.*

Avant de récupérer les données de la table, notre service distant doit au préalable se connecter à la base MySQL.

Nous ajoutons la connexion à la base au sein du constructeur du service distant :

```

<?php
class Echanges
{
    function Echanges ( )
    {
        // connexion au serveur MySQL
        mysql_connect ("localhost", "thibault", "20061982");
        // sélection de la base
        mysql_select_db ("maBase");
    }
}
?>

```

Puis nous ajoutons une méthode `recupereMenu` :

```

<?php
class Echanges

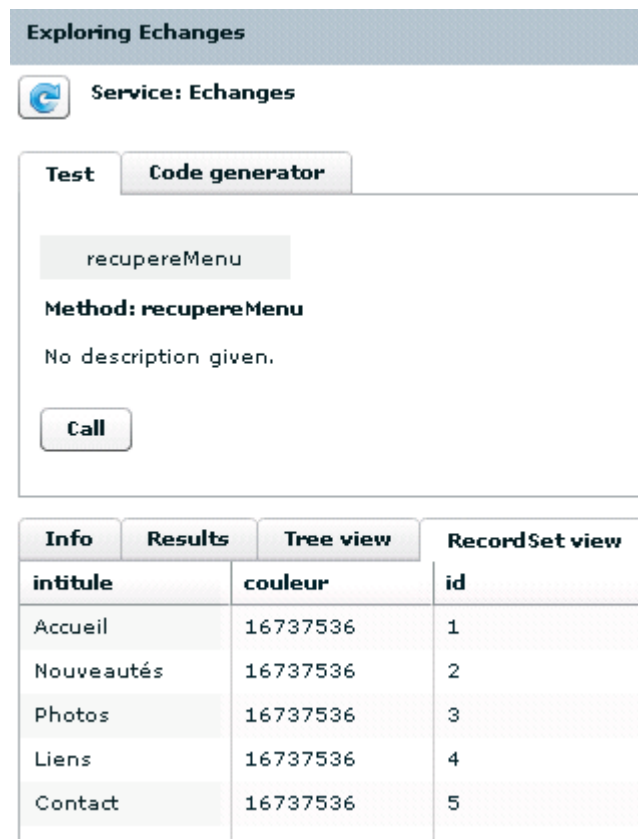
```

```
{  
  
    function Echanges ( )  
  
    {  
  
        // connexion au serveur MySQL  
        mysql_connect ("localhost", "bobgroove", "20061982");  
        // sélection de la base  
        mysql_select_db ("maBase");  
  
    }  
  
    function recupereMenu ( )  
  
    {  
  
        return mysql_query ("SELECT * FROM menu");  
  
    }  
  
}  
  
?>
```

En nous rendant auprès de l'explorateur de services nous pouvons tester directement la méthode `recupereMenu`.

A l'aide de l'onglet *RecordSet view* voyons directement au sein d'une liste le résultat de notre requête.

La figure 19-16 illustre le résultat :



*Figure 19-16. Aperçu en direct du résultat de la requête.*

Chaque champ de colonne correspond au champ de la table `menu`, grâce à ce mécanisme les données sont présentées en une seule ligne de code PHP.

Dans un nouveau document Flash nous associons la classe de document suivante :

```
package org.bytearray.document
{
    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.SecurityErrorEvent;
    import flash.events.AsyncErrorEvent;
    import flash.events.NetStatusEvent;
    import flash.net.Responder;
    import flash.net.NetConnection;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Document extends ApplicationDefault
    {
        private var connexion:NetConnection;
```

```
private var retourServeur:Responder;

// adresse de la passerelle AMFPHP
private static const PASSERELLE:String =
"http://localhost/echanges/gateway.php";

public function Document ()
{
    //création de la connexion
    connexion = new NetConnection();

    // création de l'objet de gestion des retours serveur
    retourServeur = new Responder ( succes, echec );

    // écoute des différents événements
    connexion.addEventListener( NetStatusEvent.NET_STATUS,
erreurConnexion );
    connexion.addEventListener( IOErrorEvent.IO_ERROR,
erreurConnexion );
    connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion );
    connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
erreurConnexion );

    // connexion à la passerelle
    connexion.connect ( Document.PASSERELLE );

    // appel de la méthode distante recupereMenu
    connexion.call ( "org.bytearray.test.Echanges.recupereMenu",
retourServeur );
}

private function erreurConnexion ( pEvt:Event ):void
{
    trace( pEvt );
}

private function succes ( pRetour:* ):void
{
    // affiche : [object Object]
    trace( pRetour );
}

private function echec ( pErreur:* ):void
{
    trace( pErreur.description );
}
}
```

```
| }
```

Dès l'initialisation de l'application nous appelons la méthode distante `recupereMenu`. AMFPHP retourne les données issues de la requête `MySQL` sous la forme d'un objet couramment appelé jeu d'enregistrements (*RecordSet*).

Un objet `RecordSet` possède une propriété `serverInfo` contenant les propriétés suivantes :

- `totalCount` : le nombre d'enregistrements renvoyés.
- `columnNames` : un tableau contenant le nom des champs de la table.
- `initialData` : un tableau de tableaux, contenant les données de la table.
- `Id` : identifiant de la session en cours créée par AMFPHP.
- `Version` : la version de l'objet `RecordSet`.
- `cursor` : point de départ de la lecture des données.
- `serviceName` : nom du service distant.

Si nous itérons au sein de l'objet `serverInfo` nous découvrons chacune des propriétés et leur valeurs :

```
private function succes ( pRetour:* ):void
{
    /* affiche :
       serviceName : PageAbleResult
       columnNames : id,intitule,couleur
       id : 952b03b6b26e39ff52418985866801ab
       initialData :
1,Accueil,16737536,2,Nouveautés,16737536,3,Photos,16737536,4,Liens,16737536,5,C
ontact,16737536
       totalCount : 5
       version : 1
       cursor : 1
    */
    for ( var p in pRetour.serverInfo )
    {
        trace( p, " : " + pRetour.serverInfo[p] );
    }
}
```

En utilisant AMFPHP à l'aide du framework Flex ou AIR, les ressources `MySQL` sont converties sous la forme d'objet `ArrayCollection` permettant un accès facilité aux données.

Au sein de Flash CS3, aucune classe n'est prévue pour gérer l'interprétation des `RecordSet`, nous allons donc devoir développer une petite fonction de reorganisation des données.

Nous ajoutons une méthode `conversion` :

```
private function conversion ( pSource:Object ):Array
{
    var donnees:Array = new Array();
    var element:Object;

    for ( var p:String in pSource.initialData )
    {
        element = new Object();

        for ( var q:String in pSource.columnNames )
        {
            element[pSource.columnNames[q]] = pSource.initialData[p][q];
        }

        donnees.push ( element );
    }

    return donnees;
}
```

Lorsque les données sont chargées nous construisons un tableau d'objets à l'aide de la méthode `conversion` :

```
private function succes ( pRetour:* ):void
{
    // le RecordSet est converti en tableau d'objets
    var donnees:Array = conversion ( pRetour.serverInfo );
}
```

L'idéal étant d'isoler cette méthode afin de pouvoir la réutiliser à tout moment, dans n'importe quel projet. Nous pourrions imaginer une méthode `conversion` au sein d'une classe `OutilsRemoting`.

Nous allons à présent intégrer le menu développé au cours du chapitre 7. Assurez vous d'avoir bien importé le bouton que nous avons créé associé à la classe `Bouton` :

```
package org.bytearray.document
{

```

```
import flash.display.Sprite;
import flash.events.Event;
import flash.events.IOErrorEvent;
import flash.events.SecurityErrorEvent;
import flash.events.AsyncErrorEvent;
import flash.events.NetStatusEvent;
import flash.events.MouseEvent;
import flash.net.Responder;
import flash.net.NetConnection;
import fl.transitions.Tween;
import fl.transitions.easing.Elastic;
import org.bytearray.abstrait.ApplicationDefault;

public class Document extends ApplicationDefault
{
    private var connexion:NetConnection;
    private var retourServeur:Responder;
    private var conteneurMenu:Sprite;

    // adresse de la passerelle AMFPHP
    private static const PASSERELLE:String =
"http://localhost/echanges/gateway.php";

    public function Document ()
    {
        conteneurMenu = new Sprite();

        conteneurMenu.x = 140;
        conteneurMenu.y = 120;

        addChild ( conteneurMenu );

        conteneurMenu.addEventListener ( MouseEvent.ROLL_OVER,
survolBouton, true );
        conteneurMenu.addEventListener ( MouseEvent.ROLL_OUT,
quitteBouton, true );

        //création de la connexion
        connexion = new NetConnection();

        // création de l'objet de gestion des retours serveur
        retourServeur = new Responder ( succes, echec );

        // écoute des différents événements
        connexion.addEventListener( NetStatusEvent.NET_STATUS,
erreurConnexion );
        connexion.addEventListener( IOErrorEvent.IO_ERROR,
erreurConnexion );
        connexion.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion );
        connexion.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
erreurConnexion );

        // connexion à la passerelle
        connexion.connect ( Document.PASSERELLE );

        // appel de la méthode distante recupereMenu
```



```

        connexion.call ( "org.bytearray.test.Echanges.recupereMenu",
retourServeur );

    }

    private function erreurConnexion ( pEvt:Event ):void

    {

        trace( pEvt );

    }

    private function succes ( pRetour:* ):void

    {

        // le RecordSet est converti en tableau d'objets
        var donnees:Array = filtre ( pRetour.serverInfo );

        var lng:int = donnees.length;
        var monBouton:Bouton;

        var angle:int = 360 / lng;

        for ( var i:int = 0; i< lng; i++ )

        {

            // instantiation du symbole Bouton
            monBouton = new Bouton();

            // activation du comportement bouton
            monBouton.buttonMode = true;

            // désactivation des objets enfants
            monBouton.mouseChildren = false;

            // affectation du contenu
            monBouton.maLegende.text = donnees[i].intitule;

            // disposition des occurrences
            monBouton.tween = new Tween ( monBouton, "rotation",
Elastic.easeOut, 0, angle * (i+1), 2, true );

            // on crée un objet Tween pour les effets de survol
            monBouton.tweenSurvol = new Tween ( monBouton.fondBouton,
"scaleX", Elastic.easeOut, 1, 1, 2, true );

            // ajout à la liste d'affichage
            conteneurMenu.addChild ( monBouton );

        }

    }

    private function echec ( pErreur:* ):void

    {

        trace( pErreur.description );

    }

```

```
    }

    function survolBouton ( pEvt:MouseEvent ):void
    {
        var monTween:Tween = pEvt.target.tweenSurvol;
        monTween.continueTo (1.1, 2);
    }

    function quitteBouton ( pEvt:MouseEvent ):void
    {
        var monTween:Tween = pEvt.target.tweenSurvol;
        monTween.continueTo (1, 2);
    }

    private function filtre ( pSource:Object ):Array
    {
        var donnees:Array = new Array();
        var element:Object;

        for ( var p:String in pSource.initialData )
        {
            element = new Object();

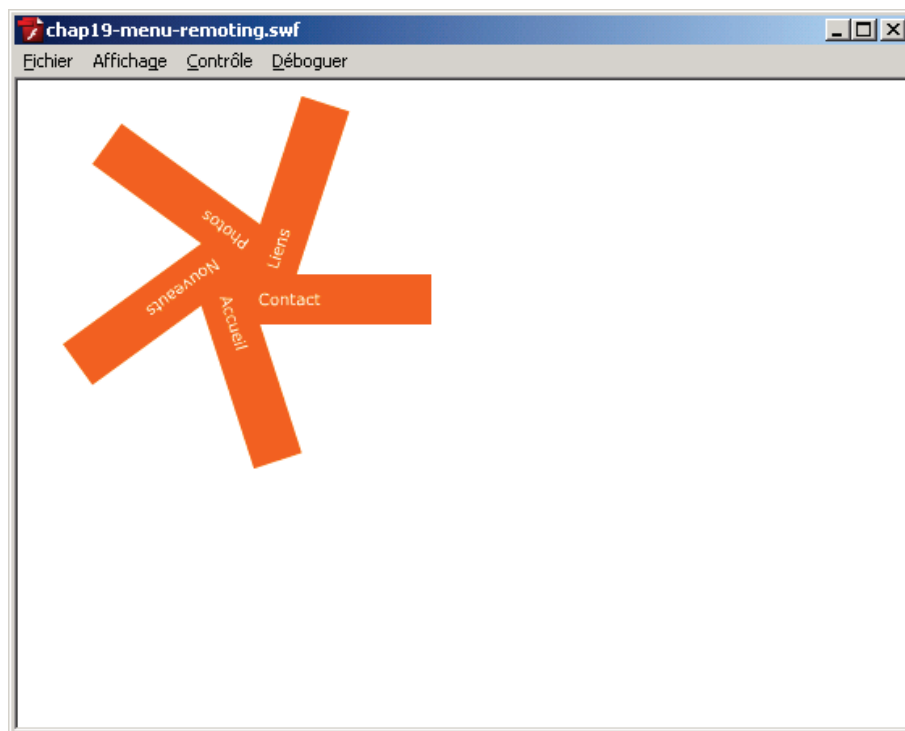
            for ( var q:String in pSource.columnNames )
            {
                element[pSource.columnNames[q]] =
pSource.initialData[p][q];
            }

            donnees.push ( element );
        }

        return donnees;
    }
}
```

En testant le code précédent, nous obtenons un menu dynamique connecté à notre base de données.

La figure 19-17 illustre le résultat :



*Figure 19-17. Menu dynamique connecté.*

Nous avons un champ de la base inutilisé pour le moment au sein de notre menu. Nous allons remédier à cela en liant la couleur de chaque bouton au champ couleur associé :

```
private function succes ( pRetour:* ):void
{
    // le RecordSet est converti en tableau d'objets
    var donnees:Array = filtre ( pRetour.serverInfo );

    var lng:int = donnees.length;
    var monBouton:Bouton;
    var transformationCouleur:ColorTransform;

    var angle:int = 360 / lng;

    for ( var i:int = 0; i < lng; i++ )
    {
        // instantiation du symbole Bouton
        monBouton = new Bouton();

        // activation du comportement bouton
        monBouton.buttonMode = true;

        // désactivation des objets enfants
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = donnees[i].intitule;
```

```

// récupération de l'objet de transformation de couleur
transformationCouleur = monBouton.fondBouton.transform.colorTransform;

// affectation d'une couleur dynamique
transformationCouleur.color = donnees[i].couleur;

// mise à jour de la couleur
monBouton.fondBouton.transform.colorTransform = transformationCouleur;

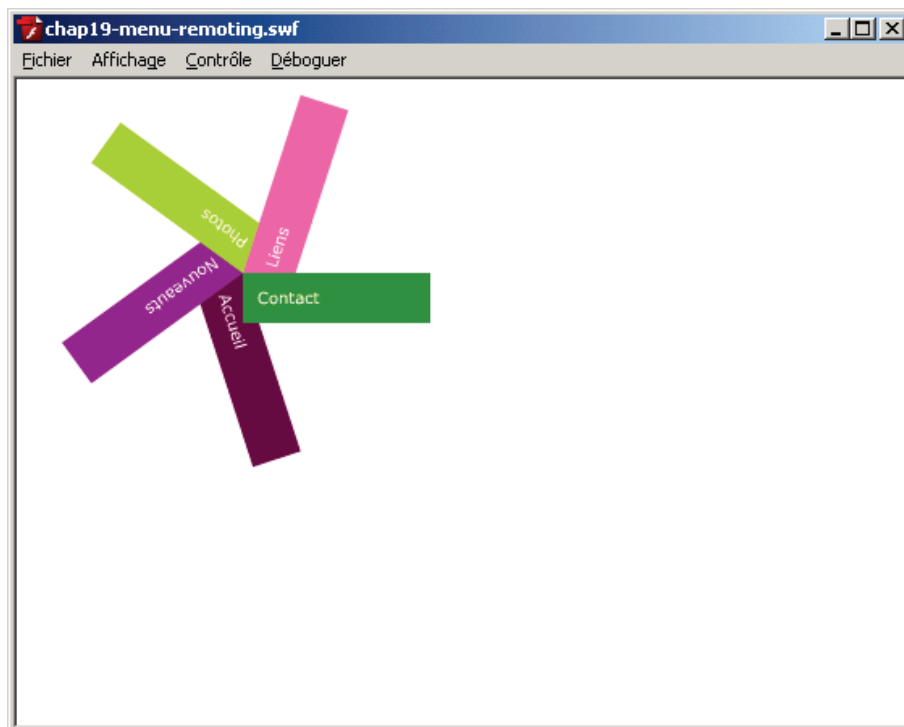
// disposition des occurrences
monBouton.tween = new Tween ( monBouton, "rotation", Elastic.easeOut,
0, angle * (i+1), 2, true );

// on crée un objet Tween pour les effets de survol
monBouton.tweenSurvol = new Tween ( monBouton.fondBouton, "scaleX",
Elastic.easeOut, 1, 1, 2, true );

// ajout à la liste d'affichage
conteneurMenu.addChild ( monBouton );
}
}

```

En modifiant les champs couleur de chaque rubrique nous obtenons le résultat illustré par la figure 19-18 :



*Figure 19-18. Menu connecté avec couleurs dynamiques.*

Grâce à Flash Remoting, l'échange de données est réellement simplifié, nous pourrions ajouter de nombreuses fonctionnalités à notre menu.

A vous de développer le gestionnaire d'administration de ce menu à l'aide de méthodes de mise à jour de la base !

### A retenir

- Flash Remoting permet de retourner directement des ressources MySQL à Flash.
- La ressource MySQL est convertie en un objet appelé RecordSet.
- Dans Flash CS3, cet objet n'est pas exploitable directement, nous devons réorganiser les données manuellement.
- Pour cela, un script peut être développé et réutilisé pour d'autres projets Flash Remoting.

## Sécurité

Une fois le développement de notre service distant terminé et notre application déployée, nous devons impérativement supprimer l'explorateur de services. Cela évite de laisser à disposition l'accès aux méthodes distantes auprès d'une personne mal intentionnée.

Pour supprimer l'explorateur de services, nous supprimons le répertoire `browser`.

Garder à l'esprit que les paquets AMF échangés peuvent être décodés très facilement. Comme nous l'avons vu précédemment, chaque paquet AMF contient de nombreuses informations.

Parmi celles-ci nous retrouvons le nom de la méthode distante à exécuter, l'adresse de la passerelle ainsi que les données à transmettre. De nombreux logiciels de déboguage tels *ServiceCapture* ou *Charles* permettent de lire ces informations.

Ces derniers sont disponibles à l'adresse suivante :

- Service Capture : <http://kevinlangdon.com/serviceCapture/>
- Charles : <http://www.xk72.com/charles/>

Une personne mal intentionnée pourrait être tentée de récupérer l'adresse de notre passerelle AMFPHP et de s'y connecter en appelant les méthodes distantes affichées par un logiciel de capture de paquets AMF tel *Service Capture*.

Afin de supprimer les éventuels messages émis par la méthode `trace` de la classe `NetDebug`, nous pouvons activer le mode « production » d'AMFPHP.

Pour activer cette fonctionnalité il convient de passer la constante `PRODUCTION_SERVER` à `true` au sein du fichier `gateway.php` :

```
| define("PRODUCTION_SERVER", true);
```

Si l'application Flash tentant d'appeler les méthodes de notre service est hébergée sur un autre serveur, l'appel de la méthode `call` de l'objet `NetConnection` entraîne une erreur de sécurité. Attention, si nous avons placé sur notre serveur un fichier de régulation autorisant tous les domaines, l'appel réussira et rendra notre service vulnérable.

Le risque peut donc venir d'une personne se connectant à notre service distant depuis l'environnement auteur de Flash, car aucune restriction de sécurité n'est appliquée dans ce cas.

Les versions 1.9 et ultérieures d'AMFPHP intègrent une nouvelle fonctionnalité permettant d'interdire toute connexion au service distant depuis le lecteur Flash autonome. L'activation de la constante `PRODUCTION_SERVER` active automatiquement ce contrôle et améliore la sécurité de notre service distant.

Pour une sécurité optimale nous pouvons utiliser la méthode `addHeader` de la classe `NetConnection` afin d'authentifier l'utilisateur en cours au sein d'AMFPHP.

Pour plus d'informations à ce sujet, rendez-vous à l'adresse suivante :

<http://www.amfphp.org/docs/authenticate.html>

## A retenir

- Une fois l'application Flash Remoting en production il faut impérativement supprimer l'explorateur de services.
- Il est fortement recommandé d'activer le mode « production » d'AMFPHP grâce à la constante `PRODUCTION_SERVER` du fichier `gateway.php`.

## La classe Service

Bien que Flash Remoting s'avère extrêmement pratique, l'utilisation de la classe `NetConnection` s'avère peu souple.

La méthode distante à appeler doit être passée sous forme de chaîne de caractères à la méthode `call` ce qui s'avère peu pratique.

Nous allons développer dans la partie suivante, une classe `Service` permettant de représenter le service distant, comme si ce dernier était un objet `ActionScript`.

En d’autres termes, la classe `Service` nous permettra de représenter sous forme d’objet `ActionScript` le service distant.

Pour appeler la méthode distante `recupereMenu`, nous écrirons :

```
// création du service coté Flash
var monService:Service = new Service();

// appel de la méthode distante
monService.recupereMenu();
```

Tous les mécanismes internes liés à l’objet `NetConnection` seront totalement transparents, rendant le développement d’applications `Flash Remoting` encore plus simple.

Au sein d’un paquetage `org.bytearray.remoting` nous définissons la classe `Service` suivante :

```
package org.bytearray.remoting
{
    import flash.events.EventDispatcher;
    import flash.events.IEventDispatcher;
    import flash.events.Event;
    import flash.utils.Proxy;
    import flash.utils.flash_proxy;

    public dynamic class Service extends Proxy implements IEventDispatcher
    {
        private var diffuseur:EventDispatcher;

        public function Service ()
        {
            diffuseur = new EventDispatcher();
        }

        override flash_proxy function hasProperty(name:*) :Boolean
        {
            return false;
        }

        override flash_proxy function getProperty(name:*) :*
        {
            return undefined;
        }

        public function toString ( ):String
```

```
        {  
            return "[object Service]";  
        }  
  
        public function addEventListener( type:String, listener:Function,  
useCapture:Boolean=false, priority:int=0, useWeakReference:Boolean=false ):void  
        {  
            diffuseur.addEventListener( type, listener, useCapture, priority,  
useWeakReference );  
        }  
  
        public function dispatchEvent( event:Event ):Boolean  
        {  
            return diffuseur.dispatchEvent( event );  
        }  
  
        public function hasEventListener( type:String ):Boolean  
        {  
            return diffuseur.hasEventListener( type );  
        }  
  
        public function removeEventListener( type:String, listener:Function,  
useCapture:Boolean=false ):void  
        {  
            diffuseur.removeEventListener( type, listener, useCapture );  
        }  
  
        public function willTrigger( type:String ):Boolean  
        {  
            return diffuseur.willTrigger( type );  
        }  
    }  
}
```

Dans un premier temps nous remarquons que la classe `Service` étend la classe `flash.utils.Proxy` et permet donc d'intercepter l'appel de méthodes inexistantes.

Le code suivant illustre le concept :

```
import org.bytearray.remoting.Service;  
  
var monService:Service = new Service();  
  
monService.methodeInexistante();
```

En testant le code précédent, l'erreur suivante est levée à l'exécution :

```
Error: Error #2090: La classe Proxy ne met pas en oeuvre callProperty. Elle  
doit être remplacée par une sous-classe.
```

Afin de pouvoir intercepter les méthodes ou appels de propriétés nous devons définir une méthode `callProperty` surchargeante :

```
override flash_proxy function callProperty ( nomMethode:*, ...parametres:* ):*  
{
```



```
        trace ( nomMethode );  
  
        return null;  
    }  
}
```

Lorsqu'une méthode est appelée sur l'instance de `Service`, la méthode `callProperty` est exécutée et renvoie en paramètre le nom de la propriété référencée.

---

Rappelez-vous qu'une méthode est considérée dans le modèle objet d'ActionScript telle une fonction référencée par une propriété. La fonction s'exécutant dans le contexte de l'instance. C'est la raison pour laquelle la méthode interceptant les appels est appelée `callProperty`.

---

En appelant à nouveau une méthode inexistante, nous voyons que la méthode interne `callProperty` est exécutée et affiche le nom de la méthode appelée :

```
import org.bytearray.remoting.Service;  
  
var monService:Service = new Service();  
  
// affiche : methodeInexistante  
monService.methodeInexistante();
```

Nous allons profiter de ce comportement pour capturer le nom de la méthode et déléguer son appel auprès de l'objet `NetConnection`.

Ainsi, pour appeler la méthode distante `envoiMessage` nous n'écrirons plus le code peu digeste suivant :

```
connexion.call ("org.bytearray.Echanges.envoiMessage", retourServeur, infos );
```

Nous préférons l'écriture simplifiée suivante :

```
monService.envoiMessage( infos );
```

Pour mettre en place ce mécanisme, un objet `NetConnection` est enveloppé au sein de l'objet `Service` afin de lui déléguer les méthodes appelées :

```
package org.bytearray.remoting  
{  
  
    import flash.events.EventDispatcher;  
    import flash.events.IEventDispatcher;  
    import flash.events.Event;  
    import flash.events.NetStatusEvent;  
    import flash.events.IOErrorEvent;  
    import flash.events.SecurityErrorEvent;  
    import flash.events.AsyncErrorEvent;  
    import flash.net.NetConnection;
```

```
import flash.utils.Proxy;
import flash.utils.flash_proxy;

public dynamic class Service extends Proxy implements IEventDispatcher
{
    private var diffuseur:EventDispatcher;
    private var connection:NetConnection;
    private var cheminService:String;
    private var passerelle:String;

    public function Service ( pService:String, pPasserelle:String,
pEncodage:int=0 )
    {
        diffuseur = new EventDispatcher();

        connection = new NetConnection();

        connection.objectEncoding = pEncodage;
        connection.client = this;
        cheminService = pService;
        passerelle = pPasserelle;

        connection.addEventListener( NetStatusEvent.NET_STATUS,
ecouteurCentralise );
        connection.addEventListener( IOErrorEvent.IO_ERROR,
ecouteurCentralise );
        connection.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
ecouteurCentralise );
        connection.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
ecouteurCentralise );

        connection.connect( passerelle );
    }

    private function ecouteurCentralise ( pEvt:Event ):void
    {
        diffuseur.dispatchEvent ( pEvt );
    }

    override flash_proxy function callProperty ( nomMethode:*,
...parametres:* ): *
    {
        trace ( nomMethode );

        return null;
    }

    override flash_proxy function hasProperty(name:*) :Boolean
    {
        return false;
    }
}
```

```
        override flash_proxy function getProperty(name:*):*
        {

            return undefined;

        }

        public function toString ( ):String
        {

            return "[object Service]";

        }

        public function addEventListener( type:String, listener:Function,
        useCapture:Boolean=false, priority:int=0, useWeakReference:Boolean=false ):void
        {
            diffuseur.addEventListener( type, listener, useCapture, priority,
        useWeakReference );
        }

        public function dispatchEvent( event:Event ):Boolean
        {
            return diffuseur.dispatchEvent( event );
        }

        public function hasEventListener( type:String ):Boolean
        {
            return diffuseur.hasEventListener( type );
        }

        public function removeEventListener( type:String, listener:Function,
        useCapture:Boolean=false ):void
        {
            diffuseur.removeEventListener( type, listener, useCapture );
        }

        public function willTrigger( type:String ):Boolean
        {
            return diffuseur.willTrigger( type );
        }

    }

}
```

Nousinstancions la classe `Service` en passant les paramètres nécessaires :

```
import org.bytearray.remoting.Service;

// création de la connexion
var monService:Service = new Service( "org.bytearray.test.Echanges",
"http://localhost/echanges/gateway.php", ObjectEncoding.AMF3 );
```

L'instance de la classe `Service` doit à présent appeler la méthode distante grâce à l'objet `NetConnection` interne.

La classe `AppelProcedure` se charge d'appeler la méthode distante :

```
package org.bytearray.remoting
{
    import flash.events.EventDispatcher;
    import flash.net.NetConnection;
    import flash.net.Responder;

    public final class ProcedureAppel extends EventDispatcher
    {
        private var recepteur:Responder;
        private var connexion:NetConnection;
        private var requete:String;
        private var parametresFinal:Array;
        private var parametres:Array;

        public function ProcedureAppel( pConnection:NetConnection,
pRequete:String, pParametres:Array )
        {
            recepteur = new Responder ( succes, echec );

            connexion = pConnection;
            requete = pRequete;
            parametres = pParametres;

        }

        private function succes ( pEvt:Object ):void
        {

        }

        private function echec ( pEvt:Object ):void
        {

        }

        internal function appel ():void
        {

            parametresFinal = new Array( requete, recepteur );

            connexion.call.apply ( connexion,
parametresFinal.concat(parametres) );

        }

    }
}
```

Afin que la procédure renseigne à notre service le résultat du serveur, nous définissons deux classes événementielles permettant de faire transiter les résultats.

La classe `EvenementResultat` est définie au sein du paquetage `org.bytearray.remoting.evenements` :

```
package org.bytearray.remoting.evenements
{
    import flash.events.Event;
```

```
public final class EvenementResultat extends Event
{
    public var resultat:Object;

    public static const RESULTAT:String = "resultat";

    public function EvenementResultat (pType:String, pDonnees:Object)
    {
        super (pType, false, false);

        resultat = pDonnees;
    }
}
```

Nous définissons une classe `EvenementErreur` au sein du même paquetage :

```
package org.bytearray.remoting.evenements
{
    import flash.events.Event;

    public final class EvenementErreur extends Event
    {
        public var erreur:Object;

        public static const ERREUR:String = "erreur";

        public function EvenementErreur (pType:String, pFault:Object)
        {
            super (pType, false, false);

            erreur = pFault;
        }
    }
}
```

La classe `ProcedureAppel` est modifiée afin de diffuser deux événements `EvenementResultat.RESULTAT` et `EvenementErreur.ERREUR` :

```
package org.bytearray.remoting
{
    import flash.events.EventDispatcher;
    import flash.net.NetConnection;
    import flash.net.Responder;

    import org.bytearray.remoting.evenements.EvenementResultat;
    import org.bytearray.remoting.evenements.EvenementErreur;

    public final class ProcedureAppel extends EventDispatcher
    {
```

```
private var recepteur:Responder;
private var connexion:NetConnection;
private var requete:String;
private var parametresFinal:Array;
private var parametres:Array;

public function ProcedureAppel( pConnection:NetConnection,
pRequete:String, pParametres:Array )
{

    recepteur = new Responder ( succes, echec );

    connexion = pConnection;
    requete = pRequete;
    parametres = pParametres;

}

private function succes ( pEvt:Object ):void
{

    dispatchEvent ( new EvenementResultat (
EvenementResultat.RESULTAT, pEvt ) );

}

private function echec ( pEvt:Object ):void
{

    dispatchEvent ( new EvenementErreur ( EvenementErreur.ERREUR,
pEvt ) );

}

internal function appel ():void
{

    parametresFinal = new Array( requete, recepteur );

    connexion.call.apply ( connexion,
parametresFinal.concat(parametres) );

}

}
```

La classe `Service` instancie donc un objet `ProcedureAppel` à chaque appel de méthode et appelle la méthode `appel` :

```
override flash_proxy function callProperty ( nomMethode:*, ...parametres:* ):*
{

    appelEnCours = cheminService + "." + nomMethode;

    procedureAppel = new ProcedureAppel ( connexion, appelEnCours,
parametres );

    procedureAppel.appel();

    return procedureAppel;

}
```

```
}
```

Lorsque nous appelons une méthode de l'objet `Service`, celui-ci délègue l'appel à l'objet `NetConnection` interne, la méthode est appelée grâce à l'instance de `ProcedureAppel`.

Nous retournons celle-ci afin de pouvoir écouter l'événement `EvenementResultat.RESULTAT` et `EvenementErreur.ERREUR`.

Nous pouvons à présent intégrer la classe `Service` à notre menu dynamique créé auparavant :

```
package org.bytearray.document

{

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.SecurityErrorEvent;
    import flash.events.AsyncErrorEvent;
    import flash.events.NetStatusEvent;
    import flash.events.MouseEvent;
    import flash.geom.ColorTransform;
    import flash.net.Responder;
    import flash.net.NetConnection;
    import flash.net.ObjectEncoding;
    import fl.transitions.Tween;
    import fl.transitions.easing.Elastic;
    import org.bytearray.abstrait.ApplicationDefault;
    import org.bytearray.remoting.Service;
    import org.bytearray.remoting.ProcedureAppel;
    import org.bytearray.remoting.evenements.EvenementResultat;
    import org.bytearray.remoting.evenements.EvenementErreur;

    public class Document extends ApplicationDefault

    {

        private var service:Service;
        private var conteneurMenu:Sprite;

        // adresse de la passerelle AMFPHP
        private static const PASSERELLE:String =
            "http://www.bytearray.org/tmp/echanges/gateway.php";
        private static const SERVICE_DISTANT:String =
            "org.bytearray.test.Echanges";

        public function Document ()

        {

            conteneurMenu = new Sprite();

            conteneurMenu.x = 140;
            conteneurMenu.y = 120;

            addChild ( conteneurMenu );
```

```

        conteneurMenu.addEventListener ( MouseEvent.ROLL_OVER,
survolBouton, true );
        conteneurMenu.addEventListener ( MouseEvent.ROLL_OUT,
quitteBouton, true );

        //création de la connexion
        service = new Service( Document.PASSERELLE,
Document.SERVICE_DISTANT, ObjectEncoding.AMF3 );

        // appel de la méthode distante
        var procedureAppel:ProcedureAppel = service.recupereMenu();

        // écoute du retour du serveur
        procedureAppel.addEventListener ( EvenementResultat.RESULTAT,
succes );
        procedureAppel.addEventListener ( EvenementErreur.ERREUR, echec
);

        // écoute des différents événements
        service.addEventListener( NetStatusEvent.NET_STATUS,
erreurConnexion );
        service.addEventListener( IOErrorEvent.IO_ERROR, erreurConnexion
);
        service.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
erreurConnexion );
        service.addEventListener( AsyncErrorEvent.ASYNC_ERROR,
erreurConnexion );
    }

    private function erreurConnexion ( pEvt:Event ):void
    {
        trace( pEvt );
    }

    private function succes ( pRetour:EvenementResultat ):void
    {
        // le RecordSet est converti en tableau d'objets
        var donnees:Array = filtre ( pRetour.resultat.serverInfo );

        var lng:int = donnees.length;
        var monBouton:Bouton;
        var transformationCouleur:ColorTransform;

        var angle:int = 360 / lng;

        for ( var i:int = 0; i< lng; i++ )
        {
            // instantiation du symbole Bouton
            monBouton = new Bouton();

            // activation du comportement bouton
            monBouton.buttonMode = true;

            // désactivation des objets enfants

```



```
        monBouton.mouseChildren = false;

        // affectation du contenu
        monBouton.maLegende.text = donnees[i].intitule;

        // récupération de l'objet de transformation de couleur
        transformationCouleur =
monBouton.fondBouton.transform.colorTransform;

        // affectation d'une couleur dynamique
        transformationCouleur.color = donnees[i].couleur;

        // mise à jour de la couleur
        monBouton.fondBouton.transform.colorTransform =
transformationCouleur;

        // disposition des occurrences
        monBouton.tween = new Tween ( monBouton, "rotation",
Elastic.easeOut, 0, angle * (i+1), 2, true );

        // on crée un objet Tween pour les effets de survol
        monBouton.tweenSurvol = new Tween ( monBouton.fondBouton,
"scaleX", Elastic.easeOut, 1, 1, 2, true );

        // ajout à la liste d'affichage
        conteneurMenu.addChild ( monBouton );
    }
}

private function echec ( pErreur:EvenementErreur ):void
{
    trace( pErreur.erreur.description );
}

function survolBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;
    monTween.continueTo (1.1, 2);
}

function quitteBouton ( pEvt:MouseEvent ):void
{
    var monTween:Tween = pEvt.target.tweenSurvol;
    monTween.continueTo (1, 2);
}

private function filtre ( pSource:Object ):Array
{

```

```
        var donnees:Array = new Array();
        var element:Object;

        for ( var p:String in pSource.initialData )
        {
            element = new Object();

            for ( var q:String in pSource.columnNames )
            {
                element[pSource.columnNames[q]] =
pSource.initialData[p][q];
            }

            donnees.push ( element );
        }

        return donnees;
    }
}
```

Désormais les écouteurs de l'objet `ProcedureAppel` reçoivent un objet événementiel de type `EvenementResultat` contenant les données au sein de la propriété `resultat`.

De la même manière si une erreur lors de l'appel est détectée, l'objet `ProcedureAppel` diffuse un événement `EvenementErreur` contenant les informations liées à l'erreur au sein de sa propriété `erreur`.

La classe `Service` peut ainsi être réutilisée dans n'importe quel projet nécessitant une connexion Flash Remoting optimisée. Cet exemple illustre une des limitations de l'héritage et met en avant la notion de composition.

## A retenir

- La classe `Proxy` permet d'intercepter l'accès à une propriété.
- Grâce à la composition, nous délégons les méthodes appelées auprès de l'instance de la classe `Service` à l'objet `NetConnection` interne.

# 20

## ByteArray

<b>LE CODAGE BINAIRE .....</b>	<b>1</b>
POSITION ET POIDS DU BIT .....	4
OCTET .....	10
ORDRE DES OCTETS .....	13
<b>LA CLASSE BYTEARRAY .....</b>	<b>15</b>
METHODES DE LECTURE ET D'ECRITURE .....	16
COPIER DES OBJETS .....	28
ECRIRE DES DONNEES AU FORMAT TEXTE .....	30
LIRE DES DONNEES AU FORMAT TEXTE .....	31
<b>COMPRESSER DES DONNEES .....</b>	<b>34</b>
<b>SAUVEGARDER UN FLUX BINAIRE .....</b>	<b>38</b>
<b>GENERER UN PDF .....</b>	<b>40</b>

### Le codage binaire

Il faut revenir à l'origine de l'informatique afin de comprendre les raisons de l'existence de la notation binaire.

La plus petite unité de mesure en informatique est représentée par les chiffres 0 et 1 définissant un état au sein d'un circuit électrique :

- 0 : le circuit est fermé.
- 1 : le circuit est ouvert.

Les processeurs équipant les ordinateurs ne comprennent donc que la notation binaire. Rassurez-vous, bien que tout cela puisse paraître compliqué dans un premier temps, il s'agit simplement d'une notation différente pour exprimer des données.

Avant de s'intéresser au fonctionnement de la classe `ByteArray`, nous devons tout d'abord être à l'aise avec le concept de *base arithmétique*.

De par l'histoire, l'homme compte en base 10 car ce dernier possède 10 doigts, c'est ce que nous appelons la *base décimale*. Pour exprimer la notion de temps, nous utilisons une *base sexagésimale* composée de 60 symboles. Ainsi, lorsque 60 secondes se sont écoulées, nous ne passons pas à 61 secondes, nous ajoutons une nouvelle unité, c'est-à-dire une minute et revenons à 0 en nombre de secondes.

Nous utilisons dans la vie de tous les jours les 10 symboles suivant pour représenter les nombres :

| 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Au delà de 9 nous devons donc combiner les symboles précédents.

Pour cela, nous ajoutons une nouvelle unité, puis nous repartons de 0. A chaque déplacement sur la gauche, nous ajoutons une puissance de 10.

La figure 20-1 illustre comment décomposer un nombre en différents groupes de puissances de 10 :

$$\begin{array}{ccc} 7 & 5 & 0 \\ 10^2 & 10^1 & 10^0 \end{array}$$

*Figure 20-1. Groupes de puissance de 10.*

Notre système décimal fonctionne par puissance de 10, nous pouvons donc exprimer le nombre 750 de la manière suivante :

$$| 7 * 100 + 5 * 10 = 7 * 10^2 + 5 * 10^1$$

Contrairement à la notation décimale, la notation binaire autorise l'utilisation des symboles 0 et 1 seulement.

Le nombre 150 s'exprime en binaire de la manière suivante :

$$| 10010110$$

Nous allons apprendre à convertir la notation binaire en notation décimale. Pour cela, nous appliquons le même concept que pour la base décimale en travaillant cette fois par puissance de 2.

La figure 20-2 illustre comment décomposer un nombre exprimé sous forme binaire en différents groupes :

1 0 0 1 0 1 1 0

$2^7$   $2^6$   $2^5$   $2^4$   $2^3$   $2^2$   $2^1$   $2^0$

*Figure 20-2. Groupes de puissance de 2.*

Très jeune, nous avons appris à compter jusqu'à 10 en base décimale. Si la norme avait été l'utilisation de la base binaire, nous aurions mémorisé la colonne de droite du tableau suivant :

Base décimale	Base binaire
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

*Tableau 1. Notation binaire.*

Afin de bien comprendre la notation binaire, il convient de s'attarder sur le concept de poids du bit. Une fois cette notion intégrée, nous introduirons la notion d'octet.

**A retenir**

- La notation binaire permet à l'origine de représenter un état au sein d'un circuit.
- Un processeur ne comprend que la notation binaire.
- Parmi les bases les plus courantes, nous comptons les bases décimales (10) et sexagésimales (60).
- La base 2 est appelée base binaire.

## Position et poids du bit

Prenons le cas du nombre 150, que nous pouvons représenter de la même manière sous forme binaire :

| 10010110

Les symboles utilisés en notation binaire sont appelés *bit*, le terme provenant de l'Anglais *binary digit*.

A l'inverse de la base décimale, où chaque déplacement sur la gauche incrémente d'une puissance de 10. En notation binaire, chaque déplacement du bit sur la gauche, augmente d'une puissance de 2.

Nous exprimons la position de chaque bit composant l'expression sous forme de poids. Le bit de poids le plus fort (*most significant bit ou msb*) est toujours positionné à l'extrême gauche, à l'inverse le bit le plus faible (*less significant bit ou lsb*) est positionné à l'extrême droite.

La figure 20-3 illustre l'emplacement du bit le plus fort :

10010110

^

bit de poids le plus fort (msb)

*Figure 20-3. Bit le plus fort.*

La figure 20-4 illustre l'emplacement du bit le plus faible :

10010110

^

bit de poids le plus faible (lsb)

*Figure 20-4. Bit le plus faible.*

Plus le poids d'un bit est fort, plus sa modification provoque un changement important du nombre. En continuant avec le nombre 150 nous voyons que si nous passons le bit le plus fort à 0 nous soustrayons  $2^7$  (128) au nombre 150 :

$$\begin{array}{r} 10010110 = 150 \\ \vee \\ 00010110 = 22 \end{array}$$

*Figure 20-5. Modification du bit le plus fort.*

A l'inverse, si nous modifions un bit de poids plus faible, la répercussion est moindre sur le nombre :

$$\begin{array}{r} 10010110 = 150 \\ \vee \\ 10010010 = 146 \end{array}$$

*Figure 20-6. Modification d'un bit de poids plus faible.*

Afin de convertir la notation binaire en notation décimale, nous multiplions chaque bit par son poids et additionnons le résultat :

$$1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$$

Soit, sous une forme plus compacte :

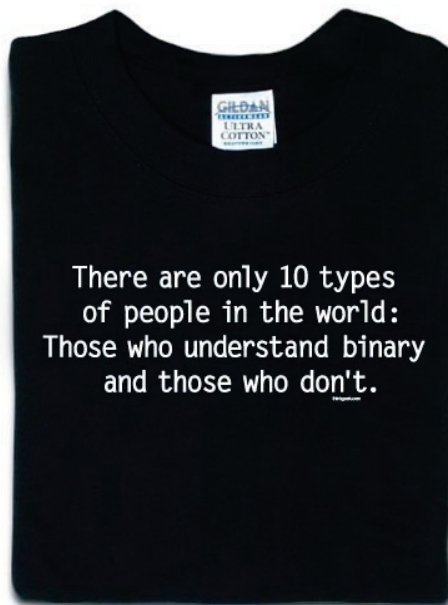
$$2^7 + 2^4 + 2^2 + 2^1 = 150$$

Nous retrouvons la notion de bits au sein des couleurs. Nous avons vu lors du chapitre 12 intitulé *Programmation Bitmap* que les couleurs étaient généralement codées en 8 bit, 16 bit ou 32 bit.

Une blague d'informaticiens consiste à dire qu'il existe 10 types de personnes dans le monde. Ceux qui comprennent le binaire et ceux qui ne le comprennent pas.

Nous savons que 10 en binaire correspond à  $2^1$  soit la valeur 2.

Vous comprendrez donc sans problème la blague inscrite sur ce fameux t-shirt illustré par la figure 20-7 :



*Figure 20-7. Humour et notation binaire.*

Comme pour la base 2, d'autres bases existent comme la base 16 appelée plus couramment *base hexadécimale*. Celle-ci est généralement utilisée pour représenter les couleurs. Nous avons déjà abordée cette notation au cours du chapitre 12 intitulé *Programmation bitmap*. Contrairement à la notation binaire composée de 2 symboles, la base 16 est composée de 16 symboles.

Dans le code suivant nous évaluons la valeur 120 en base 16 :

```
var entier:int = 120;  
// affiche : 78  
trace ( entier.toString( 16 ) );
```

A l'inverse de la base 2, ou de la base 10, la base 16 utilise 16 symboles allant de 0 à F afin d'exprimer un nombre. Nous travaillons donc non plus en puissance de 2 ou 10 mais 16.

La figure 20-8 illustre l'idée :

$$\begin{array}{c} 1 \quad A \\ 16^1 \quad 16^0 \end{array}$$

*Figure 20-8. Groupes de puissance de 16.*

Le tableau suivant regroupe les symboles utilisés en base 16 :



Base hexadécimale	Base décimale
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

*Tableau 2. Notation hexadécimale.*

La valeur 1A peut donc être exprimée de la manière suivante :

$$1 \times 16^1 + 10 \times 16^0 = 26$$

Lorsque nous travaillons avec un flux binaire, il peut être nécessaire d'exprimer un nombre sous différentes notations. Au lieu d'effectuer la conversion manuellement, nous pouvons utiliser la méthode

`toString` de la classe `Number`. Celle-ci accepte en paramètre la base dans laquelle convertir le nombre.

Dans le code suivant nous exprimons le nombre décimal 5 en notation binaire :

```
var entier:int = 5;

// affiche : 101
trace ( entier.toString( 2 ) );
```

En comptant de 1 à 10 en binaire nous retrouvons les valeurs du tableau 3 précédent :

```
var zero:int = 0;
var un:int = 1;
var deux:int = 2;
var trois:int = 3;
var quatre:int = 4;
var cinq:int = 5;
var six:int = 6;
var sept:int = 7;
var huit:int = 8;
var neuf:int = 9;
var dix:int = 10;

// affiche : 0
trace( zero.toString( 2 ) );

// affiche : 1
trace( un.toString( 2 ) );

// affiche : 10
trace( deux.toString( 2 ) );

// affiche : 11
trace( trois.toString( 2 ) );

// affiche : 100
trace( quatre.toString( 2 ) );

// affiche : 101
trace( cinq.toString( 2 ) );

// affiche : 110
trace( six.toString( 2 ) );

// affiche : 111
trace( sept.toString( 2 ) );

// affiche : 1000
trace( huit.toString( 2 ) );

// affiche : 1001
trace( neuf.toString( 2 ) );

// affiche : 1010
trace( dix.toString( 2 ) );
```

De la même manière, nous pouvons convertir un nombre en base décimale en notation hexadécimale :

```
var zero:int = 0;
var un:int = 1;
var deux:int = 2;
var trois:int = 3;
var quatre:int = 4;
var cinq:int = 5;
var six:int = 6;
var sept:int = 7;
var huit:int = 8;
var neuf:int = 9;
var dix:int = 10;
var onze:int = 11;
var douze:int = 12;
var treize:int = 13;
var quatorze:int = 14;
var quinze:int = 15;

// affiche : 0
trace( zero.toString( 16 ) );

// affiche : 1
trace( un.toString( 16 ) );

// affiche : 2
trace( deux.toString( 16 ) );

// affiche : 3
trace( trois.toString( 16 ) );

// affiche : 4
trace( quatre.toString( 16 ) );

// affiche : 5
trace( cinq.toString( 16 ) );

// affiche : 6
trace( six.toString( 16 ) );

// affiche : 7
trace( sept.toString( 16 ) );

// affiche : 8
trace( huit.toString( 16 ) );

// affiche : 9
trace( neuf.toString( 16 ) );

// affiche : a
trace( dix.toString( 16 ) );

// affiche : b
trace( onze.toString( 16 ) );

// affiche : c
trace( douze.toString( 16 ) );

// affiche : d
trace( treize.toString( 16 ) );
```

```
// affiche : e
trace( quatorze.toString( 16 ) );

// affiche : f
trace( quinze.toString( 16 ) );
```

Pour des raisons pratiques, les ingénieurs décidèrent alors de grouper les bits par paquets, c’est ainsi que naquit la notion d’octet.

## A retenir

- On parle de poids du bit pour exprimer sa puissance.
- Le bit de poids le plus fort est toujours positionné à l’extrême gauche.
- Le bit de poids le plus faible est toujours positionné à l’extrême droite.
- La méthode `toString` de la classe `Number` permet d’exprimer un nombre dans une base différente.

## Octet

L’octet permet d’exprimer une quantité de données. Nous l’utilisons tous les jours dans le monde de l’informatique pour indiquer par exemple le poids d’un fichier.

Bien entendu, nous ne dirons pas qu’un fichier MP3 pèse 3145728 octets mais plutôt 3 méga-octets. Nous ajoutons donc généralement un préfixe comme *kilo* ou *méga* permettant d’exprimer un volume d’octets :

- 1 kilooctet (ko) =  $10^3$  octets (1 000 octets)
- 1 mégaoctet (Mo) =  $10^6$  octets = 1 000 ko (1 000 000 octets)

Attention à ne pas confondre le terme de *bit* et *byte* :

1 octet = 8 bit

Un octet est donc composé de 8 bit :

| 11111111

Ce dernier peut contenir un entier naturel compris entre 0 et 255 lorsque celui-ci est dit *non-signé* (non négatif). Afin de convertir cette notation binaire sous forme décimale, nous utilisons la technique abordée précédemment.

Nous multiplions chaque bit par son poids et additionnons le résultat :

|  $1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0$

Ce qui nous donne le résultat suivant :

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Nous verrons qu'il est aussi possible d'exprimer au sein d'un octet une valeur oscillant entre -128 et 127 à l'aide d'un octet dit *signé* (négatif).

---

Il est important de noter que dans un contexte de manipulation de données binaire la base 16 est très souvent utilisée au sein de logiciels tels les éditeurs hexadécimaux afin d'optimiser la représentation d'un octet.

---

Voici une liste non exhaustive de quelques éditeurs hexadécimaux gratuits ou payants :

- Free Hex Editor Neo (gratuit)  
<http://www.hhdsoftware.com/Family/hex-editor.html>
- HxD (gratuit)  
<http://mh-nexus.de/hxd/>
- Hex Workshop (payant)  
<http://www.hexworkshop.com>
- Hexprobe (payant)  
<http://www.hexprobe.com/hexprobe>

Il est plus facile de lire la valeur d'un octet sous la notation hexadécimale suivante :

4E

Que sous une notation binaire :

1001110

Un octet non signé d'une valeur de 255 peut donc être exprimé par deux symboles seulement en notation hexadécimale :

FF

En additionnant le poids de chaque symbole et en additionnant le résultat nous obtenons la valeur 255 :

$$15 \cdot 16^1 + 15 \cdot 16^0 = 240 + 15 = 255$$

Afin de mettre cette notion en pratique, nous avons ouvert une image au format PNG au sein d'un éditeur hexadécimal.

La figure 20-9 illustre une partie des données :

Hex		
00000000:	89 50 4e 47 0d 0a 1a 0a PNG....	
00000008:	00 00 00 0d 49 48 44 52 ....IHDR	
00000010:	00 00 06 90 00 00 04 1a ... 00000018:	08 02 00 00 00 cb f5 ac .....ËÖ
00000020:	01 00 00 0f 57 69 43 43 ....WiCC	
00000028:	50 49 43 43 20 50 72 6f PICC Pro	
00000030:	66 69 6c 65 00 00 78 9c file..xœ	
00000038:	95 57 79 34 d4 7f f7 bf •Wy4Ô÷	
00000040:	9f 59 ad 63 97 2d 86 4a ŸY-c--†J	
00000048:	12 b2 17 92 5d d9 19 4b .*.']Û.K	
00000050:	c8 3e 63 19 0c 63 66 48 È>c..cfH	
00000058:	a2 10 29 ca 92 a5 85 44 ç.)Ê'¥...D	

*Figure 20-9. Editeur hexadécimal.*

Nous pouvons remarquer que chaque colonne est composée de deux symboles représentant un octet. Comme son nom l'indique, l'éditeur hexadécimal représente chaque octet en base 16 à l'aide de deux symboles.

Pourquoi un tel choix ?

Pour des raisons pratiques, car le couple de symboles FF permet de représenter la valeur maximale d'un octet, ce qui est optimisé en termes d'espaces et moins pénible à lire et mémoriser.

Comme nous le verrons plus tard, chaque fichier peut être identifié par son entête. En lisant la spécification du format PNG disponible à l'adresse suivante : <http://www.w3.org/TR/PNG/>

Nous voyons que tout fichier PNG doit obligatoirement commencer par une signature composée de cette série de valeurs décimales :

```
| 137 80 78 71 13 10 26 10
```

En convertissant en notation hexadécimale chacune de ces groupes, nous retrouvons les mêmes valeurs dans notre fichier PNG :

```
var premierOctet:int = 137;
var deuxiemeOctet:int = 80;
var troisiemeOctet:int = 78;
var quatriemeOctet:int = 71;
var cinquiemeOctet:int = 13;
var sixiemeOctet:int = 10;
var septiemeOctet:int = 26;
var huitiemeOctet:int = 10;

// affiche : 89
trace( premierOctet.toString ( 16 ) );

// affiche : 50
trace( deuxiemeOctet.toString ( 16 ) );

// affiche : 4e
```

```

trace( troisiemeOctet.toString ( 16 ) );

// affiche : 47
trace( quatriemeOctet.toString ( 16 ) );

// affiche : d
trace( cinquiemeOctet.toString ( 16 ) );

// affiche : a
trace( sixiemeOctet.toString ( 16 ) );

// affiche : 1a
trace( septiemeOctet.toString ( 16 ) );

// affiche : a
trace( huitiemeOctet.toString ( 16 ) );

```

La figure 20-10 illustre la correspondance entre chaque octet :

Hex		
00000000:	89 50 4e 47 0d 0a 1a 0a	PNG....
00000008:	00 00 00 0d 49 48 44 52	....IHDR
00000010:	00 00 06 90 00 00 04 1a	... ..
00000018:	08 02 00 00 00 cb f5 ac	....Ëö
00000020:	01 00 00 0f 57 69 43 43	....WiCC
00000028:	50 49 43 43 20 50 72 6f	PICC Pro
00000030:	66 69 6c 65 00 00 78 9c	file..xœ
00000038:	95 57 79 34 d4 7f f7 bf	•Wy4Ô÷
00000040:	9f 59 ad 63 97 2d 86 4a	ÿY-c--†J
00000048:	12 b2 17 92 5d d9 19 4b	.².'].Û.K
00000050:	c8 3e 63 19 0c 63 66 48	È>c..cfH
00000058:	a2 10 29 ca 92 a5 85 44	¢.)Ê'¥...D

Figure 20-10 Signature d'un fichier PNG.

Nous allons nous intéresser à présent à l'ordre des octets.

## A retenir

- Attention à ne pas confondre 1 octet (*byte*) et 1 bit.
- Un octet représente 8 bits.
- Un octet peut contenir une valeur maximale de 255 soit  $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$ .
- La notation hexadécimale est couramment utilisée pour représenter les octets.

## Ordre des octets

Lorsque nous devons utiliser plusieurs octets afin d'exprimer un nombre, il est important de prendre en considération l'ordre de stockage. Cette notion est appelée en Anglais *byte ordering* ou *endian nature*.

Souvenez-vous, dans la partie intitulée *Position et poids du bit* nous avons vu que le bit le plus à gauche était considéré comme celui de poids le plus fort, et inversement le bit le plus à droite était celui de poids le plus faible.

Le même concept s'applique au sein d'un groupe d'octets. Nous parlons alors d'octet le plus fort (*most significant byte* ou *MSB*) et d'octet le plus faible (*less significant byte* ou *LSB*).

---

Attention, notez que nous utilisons l'acronyme *MSB* et *LSB* en majuscule pour exprimer l'ordre des octets.

A l'inverse nous utilisons des minuscules dans les acronymes *msb* et *lsb* pour exprimer le poids d'un bit.

---

Imaginons que nous devons stocker le nombre entier suivant :

| 550000000

Ce nombre est un entier non négatif 32 bits et nécessite 4 octets afin d'être stocké, sa valeur en hexadécimale est la suivante :

| 20 C8 55 80

Les processeurs tels les *Motorola 68000* ou les processeurs *SPARC* équipant les plateformes Sun Microsystems utilise cet ordre de stockage et sont considérés comme *gros-boutiste* ou *big-endian* en Anglais.

Nous utilisons ce terme car l'octet de poids le plus fort (le plus gros) est à gauche de l'expression :

Gros-boutiste (big-endian)			
0	1	2	3
20	C8	55	80

*Tableau 3. Stockage des octets gros-boutiste.*

D'autres processeurs comme le fameux 5602 de *Motorola* ou x86 d'*Intel* sont *petit-boutiste* ou *little-endian* et stockent les octets dans le sens inverse :

Petit-boutiste (little-endian)			
0	1	2	3



80	55	C8	20
----	----	----	----

*Tableau 4. Stockage des octets petit-boutiste.*

Lors du développement d'applications ActionScript 3 utilisant la classe `ByteArray` nous devons prendre en considération cet ordre, surtout dans un contexte de communication externe.

La notion d'ordre des octets n'a pas de réelle conséquence dans le cas de lecture d'un seul octet à la fois. A l'inverse, lorsque plusieurs octets sont interprétés, nous devons absolument prendre l'ordre des octets en considération.

### A retenir

- Certaines architectures utilisent l'ordre petit-boutiste, d'autres l'ordre gros-boutiste.
- L'ordre des octets est important dans un contexte de communication entre plusieurs machines.
- Nous verrons que les classes `ByteArray`, `Socket` et `URLStream` possèdent une propriété `endian` permettant de spécifier l'ordre dans lequel stocker les octets.

## La classe ByteArray

Il est temps de mettre en application toutes les notions abordées précédemment grâce à la classe `flash.utils.ByteArray`.

Même si la classe `ByteArray` figure parmi l'une des classes les plus puissantes du lecteur Flash, une des questions les plus courantes concerne l'intérêt de pouvoir écrire un flux binaire.

L'intérêt de la classe `ByteArray` réside dans l'accès bas niveau au niveau des données. En d'autres termes, nous allons pouvoir manipuler des types de données non existants et travailler sur des octets. Cela peut nous permettre par exemple d'interpréter ou de générer n'importe quel type de fichiers en ActionScript 3.

Comme son nom l'indique, la classe `ByteArray` représente un tableau d'octets. Pour exprimer 1 kilo-octet nous utiliserons donc 1000 index du tableau.

La figure 20-11 illustre le fonctionnement d'un tableau d'octets :

[ 11111111, 11111111, ... ]

$\wedge$   
octet

$\wedge$   
octet

*Figure 20-11. Tableau d'octets.*

Comme son nom l'indique, la classe `ByteArray` possède quelques similitudes avec la classe `Array`. Les deux objets demeurent des tableaux et possèdent donc une longueur, mais chaque index composant un tableau d'octet ne peut être codé que sur 8 bits.

Voici une liste non exhaustive de quelques projets utilisant la classe `ByteArray` :

- FC64 : Emulateur Commodore 64.  
[http://codeazur.com.br/stuff/fc64\\_final/](http://codeazur.com.br/stuff/fc64_final/)
- AlivePDF : Librairie de génération de PDF.  
<http://www.alivepdf.org>
- FZip : Librairie de compression et décompression d'archives ZIP.  
<http://codeazur.com.br/lab/fzip/>
- GIF Player : Librairie de lecture de GIF animés.  
<http://www.bytearray.org/?p=95>
- GIF Player : Librairie d'encodage de GIF animés.  
<http://www.bytearray.org/?p=93>
- WiiFlash : Librairie de gestion de la manette Nintendo Wiimote.  
[www.wiiflash.org](http://www.wiiflash.org)
- Encodeurs d'images : Deux classes issues de la librairie corelib fournie par Adobe permettent l'encodage PNG et JPEG.  
<http://code.google.com/p/as3corelib/>
- Popforge Audio : Librairie de génération de sons.  
<http://www.popforge.de/>

Attention, les mêmes capacités que la classe `ByteArray` sont aussi présentes dans la classe `flash.net.Socket`. Grâce à celle-ci nous pouvons dialoguer avec l'extérieur au format brut binaire.

## Méthodes de lecture et d'écriture

Afin de créer un flux binaire nous utilisons la classe `flash.utils.ByteArray`. Bien que nous puissions créer un tableau traditionnel à l'aide de l'écriture littérale suivante :

```
| var monTableau:Array = [ ];
```

Seul le mot clé `new` permet la création d'un tableau d'octets :

```
| // création d'un tableau d'octets  
| var fluxBinaire:ByteArray = new ByteArray();
```

Pour récupérer la longueur du tableau d'octets nous utilisons sa propriété `length` :

```
| // création d'un tableau d'octets  
| var fluxBinaire:ByteArray = new ByteArray();
```

```
// affiche : 0
trace ( fluxBinaire.length );
```

Lorsqu'un tableau d'octets est créé, celui-ci est vide, de la même manière qu'un tableau traditionnel.

---

Notons qu'en ActionScript 3, la taille d'un tableau d'octets est dynamique et ne peut être fixe comme c'est le cas dans certains langages tels Java, C# ou autres.

---

Comme nous l'avons abordé précédemment, l'ordre des octets peut varier selon chaque plateforme. La propriété `endian` définie par la classe `ByteArray` permet de spécifier l'ordre des octets.

Par défaut, le tableau d'octets est *gros-boutiste* :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

// affiche : bigEndian
trace ( fluxBinaire.endian );
```

Afin de modifier l'ordre d'écriture des octets nous utiliser les constantes de la classe `flash.utils.Endian` :

- `Endian.BIG_ENDIAN` : octet le plus fort en première position.
- `Endian.LITTLE_ENDIAN` : octet le plus faible en première position.

Dans le code suivant, nous testons si l'ordre des octets est *gros-boutiste* :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

// affiche : true
trace( fluxBinaire.endian == Endian.BIG_ENDIAN );
```

Nous pouvons modifier l'ordre :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

// écriture dans l'ordre petit-boutiste
fluxBinaire.endian = Endian.LITTLE_ENDIAN;
```

Pour stocker l'entier non signé 5 au sein d'une instance de la classe `Array` nous pouvons écrire le code suivant :

```
var donnees:Array = new Array();

var nombre:uint = 5;

donnees[0] = nombre;

// affiche : 1
trace( donnees.length );
```

```
// affiche : 5
trace( donnees[0] );
```

Pour écrire le même nombre au sein d'un tableau d'octets, le nombre doit être séparé en groupe d'octets.

Ainsi, afin d'écrire un nombre entier non signé 32 bits comme c'est le cas pour le type `uint` nous devons séparer l'entier en 4 groupes de 8 bits :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture manuelle en représentation gros-boutiste
fluxBinaire[0] = (nombre & 0xFF000000) >> 24;
fluxBinaire[1] = (nombre & 0x00FF0000) >> 16;
fluxBinaire[2] = (nombre & 0x0000FF00) >> 8;
fluxBinaire[3] = nombre & 0xFF;
```

Notez que nous venons de stocker l'entier en représentation *gros-boutiste*. Une fois l'entier séparé, nous pouvons le reconstituer à l'aide des opérateurs de manipulation de bits :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture manuelle en représentation gros-boutiste
fluxBinaire[0] = (nombre & 0xFF000000) >> 24;
fluxBinaire[1] = (nombre & 0x00FF0000) >> 16;
fluxBinaire[2] = (nombre & 0x0000FF00) >> 8;
fluxBinaire[3] = nombre & 0xFF;

var nombreStocke:uint = fluxBinaire[0] << 24 | fluxBinaire[1] << 16 |
fluxBinaire[2] << 8 | fluxBinaire[3];

// affiche : 5
trace( nombreStocke );
```

Si nous devons stocker les données au format *petit-boutiste* nous devrions inverser l'ordre d'écriture :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture manuelle en représentation petit-boutiste
fluxBinaire[3] = (nombre & 0xFF000000) >> 24;
fluxBinaire[2] = (nombre & 0x00FF0000) >> 16;
fluxBinaire[1] = (nombre & 0x0000FF00) >> 8;
fluxBinaire[0] = nombre & 0xFF;

var nombreStocke:uint = fluxBinaire[3] << 24 | fluxBinaire[2] << 16 |
fluxBinaire[1] << 8 | fluxBinaire[0];
```

```
// affiche : 5
trace( nombreStocke );
```

Il s'avère extrêmement fastidieux d'écrire des données au sein d'un tableau d'octets à l'aide du code précédent.

Rassurez-vous, afin de nous faciliter la tâche le lecteur Flash va automatiquement gérer le découpage des octets ainsi que la reconstitution à l'aide de méthodes de lecture et d'écriture.

Ainsi, pour écrire un entier non signé au sein du tableau, nous utilisons la méthode `writeUnsignedInt` dont voici la signature :

```
public function writeUnsignedInt(value:int):void
```

Voici le détail du paramètre attendu :

- `value` : un entier non signé de 32 bits.

Le code fastidieux précédent peut donc être réécrit de la manière suivante :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);
```

Un entier non signé nécessite 32 bits, en testant la longueur du tableau binaire, nous voyons que 4 octets sont inscrits dans le tableau :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

// affiche : 4
trace( fluxBinaire.length );
```

Afin de lire l'entier non signé écrit, nous utilisons simplement la méthode `readUnsignedInt` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

// lève une erreur à l'exécution
var nombreStocke:uint = fluxBinaire.readUnsignedInt();
```

En testant le code précédent, l'erreur à l'exécution suivante est levée :



Pour pouvoir lire le flux, nous devons obligatoirement remettre à zéro le pointeur interne puis appeler la méthode de lecture :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStocque:uint = fluxBinaire.readUnsignedInt();

// affiche : 5
trace( nombreStocque );
```

Nous allons nous attarder sur un détail important.

Dans le code précédent, nous avons à nouveau utilisé le type `uint` afin de stocker la variable `nombre` ainsi que le résultat de la méthode `readUnsignedInt`. Tout au long de l'ouvrage, nous avons préféré l'utilisation du type `int` qui s'avère dans la plupart des cas plus rapide que le type `uint`.

Dans un contexte de manipulation de flux binaire, il convient de toujours utiliser le type lié à la méthode de lecture ou d'écriture. Nous utilisons donc le type `uint` lors de l'utilisation des méthodes `writeUnsignedInt` et `readUnsignedInt`.

Si nous ne respectons pas cette précaution, nous risquons d'obtenir des valeurs erronées. Afin de confirmer cela, nous pouvons prendre l'exemple suivant.

Dans le code suivant, nous inscrivons au sein du tableau d'octets un entier non signé d'une valeur de 3 000 000 000. En stockant le retour de la méthode `readUnsignedInt` au sein d'une variable de type `int`, nous obtenons un résultat erroné :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 3000000000;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStocque:int = fluxBinaire.readUnsignedInt();

// la machine virtuelle conserve le type à l'exécution
```

```
// affiche : -1294967296
trace( nombreStoque );
```

L'entier retourné par la méthode `readUnsignedInt` n'a pu être stocké correctement car le type `int` ne peut accueillir un entier supérieur à 2 147 483 647.

En utilisant le type approprié, nous récupérons correctement l'entier stocké :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 3000000000;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStoque:uint = fluxBinaire.readUnsignedInt();

// la machine virtuelle conserve le type à l'exécution
// affiche : 3000000000
trace( nombreStoque );
```

En analysant la figure 20-13 ci dessous, nous voyons que le nombre entier non signé d'une valeur de 3 000 000 000 occupe les 4 octets (32 bit) alloués par la machine virtuelle pour le type `uint` :

[ 0xB2, 0xD0, 0x5E, 0x00 ]

*Figure 20-13. Entier non signé au sein du tableau d'octets.*

Avez-vous remarqué que la figure précédente utilisait la notation hexadécimale afin de simplifier la représentation d'un tel nombre ?

---

Nous utiliserons désormais la notation hexadécimale  
afin de simplifier la représentation des données.

---

Nous pourrions donc exprimer une telle valeur à l'aide de la notation hexadécimale :

```
var nombre:uint = 0xB2D05E00;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);
```

A l'inverse, si nous stockons un entier non signé d'une valeur de 5 à l'aide de la méthode `writeUnsignedInt` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();
```



```
var nombre:uint = 5;

// écriture d'un entier de 32 bit non signé
fluxBinaire.writeUnsignedInt(nombre);

fluxBinaire.position = 0;

// lecture de l'entier de 32 bit non signé
var nombreStoque:uint = fluxBinaire.readUnsignedInt();

// affiche : 5
trace( nombreStoque );
```

Comme l'illustre la figure 20-14, seuls les 8 bits de poids le plus faible suffisent :

[ 0x00, 0x00, 0x00, 0x05 ]

*Figure 20-14. Le nombre 5 au sein du tableau d'octets.*

Nous pouvons donc stocker l'entier non signé au sein d'un seul octet à l'aide de la méthode `writeByte` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un octet
fluxBinaire.writeByte(nombre);
```

L'octet est inscrit à l'index 0 du tableau, la figure 20-15 illustre l'octet sous notation hexadécimale:

[ 0x05 ]

*Figure 20-15. Un octet stocké à l'index 0.*

Une fois l'octet écrit, nous pouvons le lire au sein du tableau sous la forme d'un entier non signé à l'aide la méthode `readUnsignedByte` :

```
// création d'un tableau d'octets
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 5;

// écriture d'un octet
fluxBinaire.writeByte(nombre);

fluxBinaire.position = 0;

// lecture de l'octet non signé
var nombreStoque:uint = fluxBinaire.readUnsignedByte();

// affiche : 5
trace( nombreStoque );
```

Si nous tentons d'écrire une valeur dépassant un octet en stockage, seuls les bits inférieurs sont écrits au sein du flux.

Dans le code suivant nous évaluons la valeur 260 en binaire :

```
var nombre:uint = 260;
// affiche : 100000100
trace ( nombre.toString( 2 ) );
```

Le nombre entier non signé 260 occupe 9 bits, dont voici la représentation :

								1		0	0	0	0	0	1	0	0		
	1	2	3	4	5	6	7	8			1	2	3	4	5	6	7	8	

Comme nous l'avons vu précédemment, chaque index d'un tableau d'octets ne peut contenir qu'un seul octet, soit 8 bits. Ainsi, lorsque nous tentons d'écrire une valeur nécessitant plus de 8 bits, seuls les bits *de poids le plus faible* sont inscrits.

---

Rappelez-vous, les bits dits *de poids le plus faible* sont à droite. Les bits dits *de poids le plus fort* sont ceux situés à gauche.

---

Un octet ne pouvant contenir que 8 bit, le 9ème bit débordant est ignoré :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:uint = 260;

// écriture d'un entier non signé
fluxBinaire.writeByte(nombre);

fluxBinaire.position = 0;

// lecture de l'entier non-signé
// affiche : 4
trace ( fluxBinaire.readUnsignedByte() );
```

En convertissant l'entier non signé 4 en notation binaire :

```
// lecture de l'entier non-signé au format binaire
// affiche : 100
trace ( fluxBinaire.readUnsignedByte().toString(2) );
```

Nous retrouvons les 3 bits 6, 7 et 8 de l'octet écrit :

								1		0	0	0	0	0	1	0	0		
	1	2	3	4	5	6	7	8			1	2	3	4	5	6	7	8	

Comme nous l'avons vu précédemment, un octet peut contenir une valeur oscillant 0 et 255 lorsque celui-ci est dit non signé. A l'inverse un octet signé, peut contenir un entier relatif allant de -128 à 127.

Ainsi si nous inscrivons un entier signé d'une valeur de -80, nous devons lire l'octet avec la méthode `readByte` :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:int = -80;

// écriture d'un entier signé
fluxBinaire.writeByte(nombre);

fluxBinaire.position = 0;

// lecture de l'octet signé
// affiche : -80
trace( fluxBinaire.readByte() );
```

Nous pouvons alors nous poser la question du type à utiliser lorsque nous utilisons les méthodes `writeByte` et `readByte` et `readUnsignedByte`.

Nous avons vu précédemment qu'il était recommandé d'utiliser le type `uint` lors de l'utilisation des méthodes `writeUnsignedInt` et `readUnsignedInt`. Il n'existe pas, à l'heure d'aujourd'hui de type `byte` ou `ubyte` en ActionScript 3.

Nous utilisons donc le type `int` en remplacement qui permet de stocker toutes les valeurs possibles d'un octet signé ou non signé.

Afin de stocker une valeur supérieure à 255 nous devons travailler sur deux octets. Prenons le cas du nombre entier 1200 sous sa forme binaire :

```
      1 0 0      1 0 1 1 0 0 0 0
    | 1 2 3 4 5 6 7 8 | | 1 2 3 4 5 6 7 8 |
```

Deux octets sont nécessaires à l'écriture de ce nombre, pour cela nous utilisons la méthode `writeShort` dont voici la signature :

```
public function writeShort(value:int):void
```

Voici le détail du paramètre attendu :

- `value` : Un entier de 32 bits. Seuls les 16 bits inférieurs sont écrits dans le flux d'octets.

Un couple de deux octets est généralement appelé *mot* ou *word* en Anglais, celui-ci pouvant contenir une valeur allant de -32768 à 32767 ou 0 à 65 535 si celui est dit non signé.



```
fluxBinaire.position = 0;

// lecture du premier octet
var premierOctet:int = fluxBinaire.readUnsignedByte();

// lecture du deuxième octet
var secondOctet:int = fluxBinaire.readUnsignedByte();

// affiche : 4
trace( premierOctet );

// affiche : 176
trace( secondOctet );
```

Bien entendu, nous pouvons stocker un nombre à virgule flottante au sein du tableau. ActionScript 3 utilise la norme IEEE 754 afin de traiter les nombres à virgule flottante.

Pour cela nous utilisons la méthode `writeFloat` :

```
public function writeFloat(value:Number):void
```

Voici le détail du paramètre attendu :

- `value` : un nombre à virgule flottante de 32 bits.

Dans le code suivant, nous inscrivons un flottant de 32 bits :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:Number = 12.5;

// écriture d'un flottant de 32 bits
fluxBinaire.writeFloat(nombre);

// réinitialisation du pointeur
fluxBinaire.position = 0;

// lecture du flottant
var flottant:Number = fluxBinaire.readFloat();

// affiche : 12.5
trace( flottant );

// affiche : 1100
trace( flottant.toString(2) );
```

La méthode `writeFloat` écrit un nombre au format 32 bits, 4 octets sont donc nécessaire au stockage d'un nombre flottant :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

var nombre:Number = 12.5;

// écriture d'un flottant de 32 bits
fluxBinaire.writeFloat(nombre);

// affiche : 4
trace( fluxBinaire.length );
```

Il n'existe pas en ActionScript 3 de type `float`, nous utilisons donc le type `Number` en remplacement.

Si nous souhaitons stocker un nombre à virgule flottante plus grand, équivalent au type `Number` codé sur 64 bits nous pouvons utiliser la méthode `writeDouble` :

```
// création d'un flux binaire
var fluxBinaire:ByteArray = new ByteArray();

// écriture d'un nombre à virgule flottante
fluxBinaire.writeDouble(12.5);

// affiche : 8
trace( fluxBinaire.length );
```

Notons qu'un `double` est l'équivalent du type `Number`.

ECMAScript 4 définit un type `double`, nous pourrions donc voir ce type apparaître un jour en ActionScript 3.

## A retenir

- La classe `ByteArray` définit un ensemble de méthodes permettant d'écrire différents types de données.
- Certains types de données écrits au sein du tableau d'octets n'ont pas d'équivalent en ActionScript 3. C'est le cas des types `float`, `byte`, et `short`.
- Ces méthodes découpent automatiquement les valeurs passées en paramètres en groupes d'octets.

## Copier des objets

La classe `ByteArray` intègre un sérialiseur et désérialiseur AMF permettant l'écriture de types au format AMF.

Nous pouvons utiliser la méthode `writeObject` de manière détournée en passant un objet, celui-ci est alors inscrit au sein du flux :

```
var fluxBinaire:ByteArray = new ByteArray();

// définition d'un objet
var parametres:Object = { age : 25, nom : "Bob" };

// écriture de l'objet au sein du flux
fluxBinaire.writeObject( parametres );
```

Afin de créer une copie de ce dernier nous appelons la méthode `readObject` :

```
var fluxBinaire:ByteArray = new ByteArray();

// définition d'un objet
var parametres:Object = { age : 25, nom : "Bob" };
```

```
// écriture de l'objet au sein du flux
fluxBinaire.writeObject( parametres );

fluxBinaire.position = 0;

// copie de l'objet parametres
var copieParametres:Object = fluxBinaire.readObject();
```

Nous pouvons ainsi créer une fonction personnalisée réalisant en interne tout ce processus :

```
function copieObjet ( pObjet:* ):*
{
    var fluxBinaire:ByteArray = new ByteArray();

    fluxBinaire.writeObject( pObjet );

    fluxBinaire.position = 0;

    return fluxBinaire.readObject();
}
```

Afin de copier un objet, nous devons simplement appeler la fonction `copieObjet` :

```
// définition d'un objet
var parametres:Object = { age : 25, nom : "Bob" };

var copieParametres:Object = copieObjet ( parametres );

/* affiche :
nom   : Bob
age   : 25
*/
for ( var p:String in copieParametres ) trace( p, " : ", copieParametres[p] );
```

En modifiant les données de l'objet d'origine, nous voyons que l'objet `copieParametres` est bien dupliqué :

```
var copieParametres:Object = copieObjet ( parametres );

// modification du nom dans l'objet d'origine
parametres.nom = "Stevie";

/* affiche :
nom   : Bob
age   : 25
*/
for ( var p:String in copieParametres ) trace( p, " : ", copieParametres[p] );
```

Notez que cette technique ne fonctionne que pour les objets littéraux et non les instances de classes.

---

## A retenir

---

- La méthode `writeObject` permet d'écrire un objet composite au sein du tableau d'octets.
- Grace à la méthode `readObject` nous pouvons créer une copie de l'objet écrit.

## Ecrire des données au format texte

Afin de stocker des données au format texte nous pouvons utiliser la méthode `writeUTFBytes` dont voici la signature :

```
| public function writeUTFBytes(value:String):void
```

Chaque caractère composant la chaîne est encodé sur une suite d'un à quatre octets. Dans le code suivant, nous écrivons un texte simple :

```
| var fluxBinaire:ByteArray = new ByteArray();  
|  
| var chaine:String = "Bonjour Bob";  
|  
| // affiche : 11  
| trace( chaine.length );  
|  
| fluxBinaire.writeUTFBytes(chaine);  
|  
| // affiche : 11  
| trace( fluxBinaire.length );
```

Si nous utilisons les 127 premiers caractères de l'alphabet, nous voyons que chaque caractère est codé sur un octet. A l'aide de la méthode `readByte`, nous pouvons lire récupérer le caractère de chaque octet :

```
| var fluxBinaire:ByteArray = new ByteArray();  
|  
| var chaine:String = "Bonjour Bob";  
|  
| // affiche : 19  
| trace( chaine.length );  
|  
| fluxBinaire.writeUTFBytes(chaine);  
|  
| // affiche : 20  
| trace( fluxBinaire.length );  
|  
| fluxBinaire.position = 0;  
|  
| // affiche : 66  
| trace( fluxBinaire.readByte() );  
|  
| // affiche : 111  
| trace( fluxBinaire.readByte() );
```

A l'inverse, si nous utilisons des caractères spéciaux, ils seront alors codés sur plusieurs octets :

```
| var fluxBinaire:ByteArray = new ByteArray();  
|  
| var chaine:String = "Bonjour Bob ça va ?";
```



```
// affiche : 19
trace( chaine.length );

fluxBinaire.writeUTFBytes(chaine);

// affiche : 20
trace( fluxBinaire.length );
```

Dans le code précédent, le caractère `ç` est codé sur deux octets.

## Lire des données au format texte

Comme nous l'avons vu précédemment, il est primordial de ne pas tenter sortir du flux. Souvenez-vous, précédemment nous avons tenté de lire un octet non disponible au sein du flux, l'erreur suivante fut levée à l'exécution :

```
Error: Error #2030: Fin de fichier détectée.
```

Afin de garantir de ne jamais sortir du flux, nous utilisons la propriété `bytesAvailable`. Celle-ci renvoie le nombre d'octets disponible, autrement dit la soustraction de la longueur totale du flux et de la position du pointeur interne.

Dans le code suivant nous inscrivons des données texte au sein d'un tableau d'octets et calculons manuellement le nombre d'octets disponibles à la lecture :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// calcul manuel du nombre d'octets disponibles à la lecture
// affiche : 10
trace( fluxBinaire.length - fluxBinaire.position );
```

En réinitialisant le pointeur à l'aide de la propriété `position`, nous obtenons 10 octets disponibles.

Dans l'exemple suivant, nous utilisons la propriété `bytesAvailable`, qui permet de renvoyer automatiquement le nombre d'octets disponibles à la lecture :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// affiche : 10
```

```
| trace( fluxBinaire.bytesAvailable );
```

Nous utilisons très souvent cette propriété afin d’être sûr de ne pas aller trop loin dans la lecture des octets.

Afin d’extraire le texte stocké au sein du flux, nous utilisons la méthode `readUTFBytes` ayant la signature suivante :

```
| public function readUTFBytes(length:uint):String
```

En passant le nombre d’octets à lire, celle-ci décode automatiquement chaque octet en caractère UTF-8. Ainsi dans le code suivant, nous lisons uniquement les 3 premiers caractères du flux :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// extrait les 3 premiers caractères du flux
// affiche : Bob
trace( fluxBinaire.readUTFBytes( 3 ) );
```

Nous pouvons donc utiliser la propriété `bytesAvailable` afin d’être sûr de lire la totalité du texte stockée dans le flux :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// extrait la totalité de la chaîne de caractères
// affiche : Bob Groove
trace( fluxBinaire.readUTFBytes( fluxBinaire.bytesAvailable ) );
```

Nous utilisons généralement la propriété `bytesAvailable` avec une boucle `while` afin d’extraire chaque caractère :

```
var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte
fluxBinaire.writeUTFBytes("Bob Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

while ( fluxBinaire.bytesAvailable > 0 )
{
    /* affiche :
    B
    o
    b

```

```

G
r
o
o
v
e
*/
trace( String.fromCharCode( fluxBinaire.readByte() ) );

}

```

Nous lisons chaque octet, tant qu'il en reste à lire. Afin de trouver le caractère correspondant au nombre, nous utilisons la méthode `fromCharCode` de la classe `String`.

Si nous insérons des caractères spéciaux, l'appel de la méthode `readUTFBytes` décode correctement les caractères encodés sur plusieurs octets :

```

var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte avec des caractères spéciaux
fluxBinaire.writeUTFBytes("Bob ça Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

// extrait la totalité de la chaîne de caractères
// affiche : Bob ça Groove
trace( fluxBinaire.readUTFBytes( fluxBinaire.bytesAvailable ) );

```

A l'inverse, si nous utilisons la méthode `readByte` en évaluant chaque caractère, nous ne pourrions interpréter correctement le caractère `ç` encodé sur deux octets :

```

var fluxBinaire:ByteArray = new ByteArray();

// écriture de données texte avec des caractères spéciaux
fluxBinaire.writeUTFBytes("Bob ça Groove");

// réinitialisation du pointeur
fluxBinaire.position = 0;

while ( fluxBinaire.bytesAvailable > 0 )
{
    /* affiche :
    B
    o
    b
    ç
    a

    G
    r
    o

```

```
o
v
e
*/
trace( String.fromCharCode( fluxBinaire.readByte() ) );
}
```

La méthode `readByte` n'est pas adaptée au décodage de chaînes encodées.

## A retenir

- Afin de décoder sous forme de chaîne de caractères UTF-8 un ensemble d'octets, nous utilisons la méthode `readUTFBytes`.
- Celle-ci accepte en paramètre, le nombre d'octets à lire au sein du flux.

## Compresser des données

La classe `ByteArray` dispose d'une méthode de compression de données utilisant l'algorithme zlib dont la spécification est disponible à l'adresse suivante :

<http://www.ietf.org/rfc/rfc1950.txt>

Il peut être intéressant d'utiliser cette fonctionnalité afin de compresser des données devant être sauvegardées.

Prenons le cas d'application suivant :

Nous devons sauver sur le poste de l'utilisateur un volume de données important. Afin de faciliter la représentation de ces données, un objet XML est utilisé. Pour stocker des données sur le poste client, nous utilisons la classe `flash.net.SharedObject` permettant de créer des cookies permanents.

Dans le code suivant, nous chargeons un fichier XML d'une taille de 1,5Mo au format binaire afin d'obtenir directement un objet `ByteArray` contenant le flux XML :

```
var chargeur:URLLoader = new URLLoader();
chargeur.dataFormat = URLLoaderDataFormat.BINARY;
chargeur.load( new URLRequest ( "donnees.xml" ) );
chargeur.addEventListener( Event.COMPLETE, chargementTermine );
function chargementTermine ( pEvt:Event ):void
{
}
```

```
// accès au flux binaire XML
var fluxXML:ByteArray = pEvt.target.data;

// affiche : 1547.358
trace( fluxXML.length / 1024 );

}
```

Puis nous sauvegardons le flux XML au sein d'un cookie permanent :

```
var chargeur:URLLoader = new URLLoader();

chargeur.dataFormat = URLLoaderDataFormat.BINARY;

chargeur.load( new URLRequest ( "donnees.xml" ) );

chargeur.addEventListener( Event.COMPLETE, chargementTermine );

// crée un cookie permanent du nom de "cookie"
var monCookie:SharedObject = SharedObject.getLocal("cookie");

function chargementTermine ( pEvt:Event ):void

{

    // accès au flux binaire XML
    var fluxXML:ByteArray = pEvt.target.data;

    // affiche : 1547.358
    trace( fluxXML.length / 1000 );

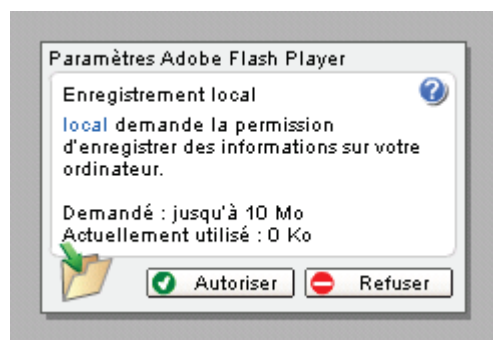
    // écriture du flux XML dans le cookie
    monCookie.data.donneesXML = fluxXML;

    // sauvegarde du cookie sur le disque dur
    monCookie.flush();

}
```

A l'appel de la méthode `flush`, le lecteur Flash tente de sauvegarder les données sur le disque. Le flux XML de large poids nécessite plus de 1 Mo d'espace disque et provoque l'ouverture du panneau *Enregistrement local* afin que l'utilisateur accepte la sauvegarde.

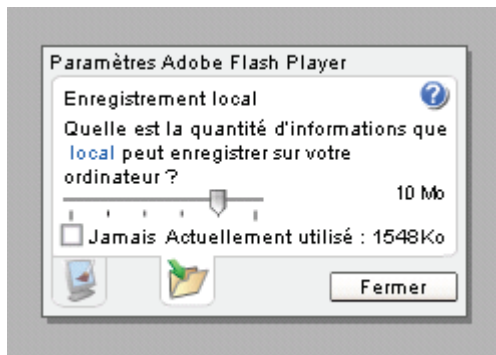
La figure 20-16 illustre le panneau :



*Figure 20-16. Panneau Enregistrement local.*

Nous remarquons que le lecteur Flash nécessite alors une autorisation de 10 mo afin de sauver le cookie. 1548 Ko sont nécessaire à la sauvegarde du cookie :

La figure 20-17 illustre l'espace actuellement utilisé par le cookie :



*Figure 20-17. Espace utilisé par le cookie permanent.*

En compressant simplement le flux XML à l'aide de la méthode `compress`, nous réduisons la taille du flux de près de 700% :

```
function chargementTermine ( pEvt:Event ):void
{
    // accès au flux binaire XML
    var fluxXML:ByteArray = pEvt.target.data;

    // affiche : 1547.358
    trace( fluxXML.length / 1024 );

    // compression du flux XML
    fluxXML.compress();

    // affiche : 212.24
    trace( fluxXML.length / 1024 );

    // écriture du flux XML dans le cookie
    monCookie.data.donneesXML = fluxXML;

    // sauvegarde du cookie sur le disque dur
    monCookie.flush();
}
```

En testant le code précédent, si nous affichons le panneau *Enregistrement local*, nous voyons que le cookie ne nécessite que 213 Ko d'espace disque :

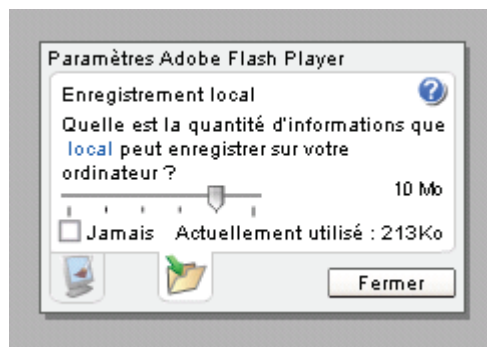


Figure 20-18. Un octet stocké à l'index 0.

Afin de lire le flux sauvegardé, nous devons appeler la méthode `uncompress`, puis extraire la chaîne compressée à l'aide de la méthode `readUTFBytes` :

```
function chargementTermine ( pEvt:Event ):void
{
    // accès au flux binaire XML
    var fluxXML:ByteArray = pEvt.target.data;

    // affiche : 1547.358
    trace( fluxXML.length / 1024 );

    fluxXML.compress();

    // affiche : 212.24
    trace( fluxXML.length / 1024 );

    // ecriture du flux XML dans le cookie
    monCookie.data.donneesXML = fluxXML;

    // sauvegarde du cookie sur le disque dur
    monCookie.flush();

    var fluxBinaireXML:ByteArray = monCookie.data.donneesXML;

    // décompression du flux XML binaire
    fluxXML.uncompress();

    // lecture de la chaîne XML
    var chaineXML:String = fluxBinaireXML.readUTFBytes (
    fluxBinaireXML.bytesAvailable );

    // reconstitution d'un objet XML
    var donneesXML:XML = new XML ( chaineXML );
}
```

Grâce à la chaîne extraite du tableau d'octets, nous récréons un objet XML utilisable. La méthode `compress` nous a permis de réduire le poids du fichier XML de près de 1335 Ko.

## A retenir

- La méthode `compress` de la classe `ByteArray` permet la compression des données binaires à l'aide de l'algorithme zlib.
- La méthode `uncompress` permet de décompresser le flux.

## Sauvegarder un flux binaire

Le lecteur Flash 9 n'a pas la capacité d'exporter un flux généré en ActionScript 3 par la classe `FileReference`. Il serait intéressant de pouvoir passer à la méthode `download` de l'objet `FileReference` un tableau d'octets afin que le lecteur nous propose de sauver le flux, mais une telle fonctionnalité n'est pas présente dans le lecteur Flash 9.

Souvenez-vous, au cours du précédent chapitre nous avons exporté une image du même type grâce à AMFPHP. Dans l'exemple suivant nous allons exporter le tableau d'octets sans utiliser AMFPHP. La première étape consiste à utiliser une classe générant un fichier spécifique telle une image, une archive zip, ou n'importe quel type de fichier.

Nous allons réutiliser la classe d'encodage `EncodeurPNG` que nous avons utilisé au cours du précédent chapitre. Rappelez-vous, la classe `EncodeurPNG` nécessite en paramètre, une image de type `BitmapData` afin d'encoder les pixels dans une enveloppe PNG.

Nous créons dans le code suivant, une image d'une dimension de 320 par 240 pixels :

```
// import de la classe EncodeurPNG
import org.bytearray.encodage.images.EncodeurPNG ;

// création d'une image
var donneesBitmap:BitmapData = new BitmapData ( 320, 240, false, 0x990000 );

// encodage au format PNG
var fluxBinairePNG:ByteArray = EncodeurPNG.encode ( donneesBitmap );

// affiche : 690
trace( fluxBinairePNG.length );
```

Nous obtenons une longueur de 690 octets. Souvenez-vous, en début de chapitre, nous avons abordé la structure d'un fichier PNG.

Pour exporter le flux nous devons transmettre par connexion HTTP le flux binaire, puis prévoir un script serveur afin de rendre le flux disponible en téléchargement par une fenêtre appropriée.

Pour permettre cela, nous créons un fichier `export.php` contenant le code PHP suivant :

```
<?php
```



```
if ( isset ( $GLOBALS["HTTP_RAW_POST_DATA"] ) ) {  
  
    $flux = $GLOBALS["HTTP_RAW_POST_DATA"];  
  
    header('Content-Type: image/png');  
    header("Content-Disposition: attachment; filename=".$_GET['nom']);  
    echo $flux;  
  
} else echo 'An error occured.';  
  
?>
```

Nous sauvons le fichier `export.php` sur notre serveur local, puis nous transmettons le flux binaire au script distant en passant le tableau d'octets à la propriété `data` de l'objet `URLRequest` :

```
// import de la classe EncodeurPNG  
import org.bytearray.encodage.images.EncodeurPNG ;  
  
// création d'une image  
var donneesBitmap:BitmapData = new BitmapData ( 320, 240, false, 0x990000 );  
  
// encodage au format PNG  
var fluxBinairePNG:ByteArray = EncodeurPNG.encode ( donneesBitmap );  
  
// entête HTTP au format brut  
var enteteHTTP:URLRequestHeader = new URLRequestHeader ( "Content-type",  
    "application/octet-stream" );  
  
var requete:URLRequest = new  
    URLRequest("http://localhost/export_image/export.php?nom=sketch.jpg");  
  
requete.requestHeaders.push(enteteHTTP);  
  
requete.method = URLRequestMethod.POST;  
  
requete.data = fluxBinairePNG;  
  
navigateToURL(requete, "_blank");
```

Notons que grâce à la classe `URLRequestHeader`, nous indiquons au lecteur Flash de ne pas traiter les données à transmettre en HTTP en tant que chaîne mais en tant que flux binaire.

---

Souvenez-vous, lors du chapitre 14 intitulé *Charger et envoyer des données*, nous avons découvert qu'il était impossible depuis l'environnement de développement de Flash, d'utiliser la méthode POST en utilisant la fonction `navigateToURL`.

Il est donc obligatoire de tester le code précédent au sein d'une page navigateur. Dans le cas contraire, le flux binaire sera transmis par la méthode GET et sera donc traité telle une chaîne de caractères.

---

Si nous avions voulu sauver l'image sur le serveur, nous aurions utilisé le code PHP suivant au sein du fichier `export.php` :

```
<?php
if ( isset ( $GLOBALS["HTTP_RAW_POST_DATA"] ) ) {

    $flux = $GLOBALS["HTTP_RAW_POST_DATA"];

    $fp = fopen($_GET['nom'], 'wb');
    fwrite($fp, $im);
    fclose($fp);

}
?>
```

Bien entendu, le code précédent peut être modifié afin de sauver le flux binaire dans un répertoire spécifique.

## A retenir

- Le lecteur Flash est incapable d'exporter un flux binaire sans un script serveur.
- Lors de l'envoi de données binaire depuis le lecteur Flash, nous devons obligatoirement utiliser la classe `URLRequestHeader` afin de préciser que les données transmises sont de type binaire.
- Les données binaires arrivent en PHP au sein de la propriété `HTTP_RAW_POST_DATA` du tableau `$GLOBALS` : `$GLOBALS["HTTP_RAW_POST_DATA"]`.

## Générer un PDF

Afin de démontrer l'intérêt de la classe `ByteArray` dans un projet réel, nous allons générer un fichier PDF dynamiquement en ActionScript 3. Pour cela, nous allons utiliser la librairie `AlivePDF` qui s'appuie sur la classe `ByteArray` pour générer le fichier PDF.

En réalité il est aussi possible de générer un fichier PDF en ActionScript 1 et 2 sans avoir recours à la classe `ByteArray`, car le format PDF est basé sur une simple chaîne de caractères. En revanche l'intégration d'images ou autres fichiers nécessite une manipulation des données binaire ce qui est réservé à ActionScript 3.

Voici le contenu d'un fichier PDF ouvert avec le bloc notes :

```
%PDF-1.4
1 0 obj
<< /Type /Catalog
/Outlines 2 0 R
/Pages 3 0 R
>>
```

```
endobj
2 0 obj
<< /Type Outlines
/Count 0
>>
endobj
3 0 obj
<< /Type /Pages
/Kids [4 0 R]
/Count 1
>>
endobj
4 0 obj
<< /Type /Page
/Parent 3 0 R
/MediaBox [0 0 612 792]
/Contents 5 0 R
/Resources << /ProcSet 6 0 R >>
>>
endobj
5 0 obj
<< /Length 35 >>
stream
...Page-marking operators...
endstream
endobj
6 0 obj
[/PDF]
endobj
xref
0 7
0000000000 65535 f
0000000009 00000 n
0000000074 00000 n
0000000120 00000 n
0000000179 00000 n
0000000300 00000 n
0000000384 00000 n
trailer
<< /Size 7
/Root 1 0 R
>>
startxref
408
%%EOF
```

Afin de générer un PDF en ActionScript nous utilisons la librairie **AlivePDF** disponible à l'adresse suivante :

<http://code.google.com/p/alivepdf/downloads/list>

Nous allons utiliser la version 0.1.4 afin de créer un PDF de deux pages contenant du texte sur la première page, puis une image sur la deuxième page.

Une fois les sources de la librairie téléchargées. Nous plaçons le répertoire **org** contenant les classes à côté d'un nouveau document FLA, puis nous importons la classe PDF :

```
import org.alivepdf.pdf.PDF;
import org.alivepdf.layout.Orientation;
import org.alivepdf.layout.Unit;
import org.alivepdf.layout.Size;
import org.alivepdf.layout.Layout;
import org.alivepdf.display.Display;
import org.alivepdf.saving.Download;
import org.alivepdf.saving.Method;

var myPDF:PDF = new PDF ( Orientation.PORTRAIT, Unit.MM, Size.A4 );
```

Nous définissons le mode d’affichage du PDF :

```
myPDF.setDisplayMode( Display.FULL_PAGE, Layout.SINGLE_PAGE );
```

Puis nous ajoutons une nouvelle page à l’aide de la méthode `addPage` :

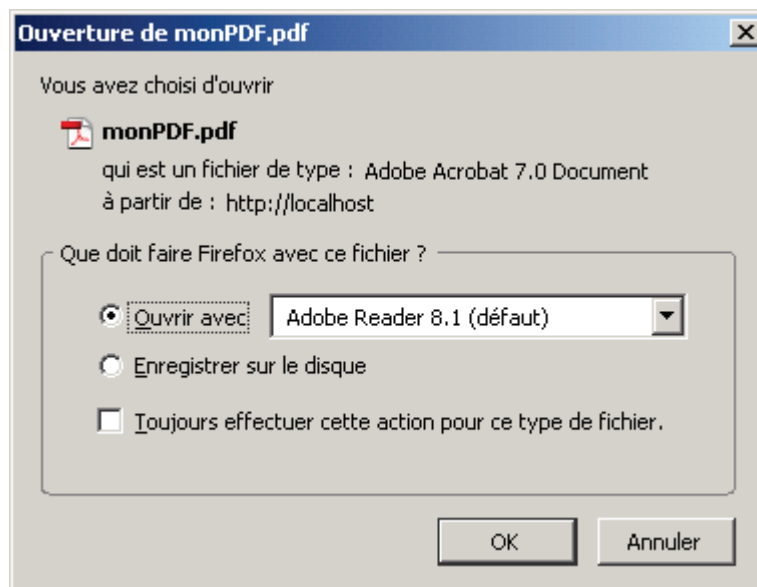
```
myPDF.addPage();
```

Enfin nous sauvons le PDF :

```
myPDF.savePDF ( Method.REMOTE, 'http://localhost/pdf/create.php',
Download.ATTACHMENT, 'monPDF.pdf' );
```

Nous devons passer en deuxième paramètre à la méthode `savePDF`, l’adresse du script `create.php` présent dans les sources du projet `AlivePDF`.

A l’exécution, le SWF ouvre la fenêtre illustrée en figure 20-19 :



*Figure 20-19. Sauvegarde du document PDF.*

A l’ouverture le PDF contient une seule page blanche. Afin d’enrichir notre document PDF, nous allons ajouter du texte, en important deux nouvelles classes :

```
import org.alivepdf.fonts.Style;
import org.alivepdf.fonts.FontFamily;
import org.alivepdf.colors.RGBColor;
```

Puis nous utilisons les différentes méthodes d'écriture de texte :

```
myPDF.textStyle ( new RGBColor ( 255, 100, 0 ) );
myPDF.setFont( FontFamily.HELVETICA, Style.BOLD );
myPDF.setFontSize ( 20 );
myPDF.addText ( 'Voilà du texte !', 70, 12);
myPDF.addLink ( 70, 4, 52, 16, "http://alivepdf bytearray.org");
```

En testant le code précédent, un document PDF est généré, à l'ouverture, le texte est affiché et redirige sur le site du projet **AlivePDF** lors du clic.

La figure 20-20 illustre le résultat :



**Voilà du texte !**

*Figure 20-20. Texte intégré au PDF.*

Afin d'intégrer une image, nous importons une image dans la bibliothèque, puis nous l'associons à une classe nommée Logo.

Grâce à la méthode `addImage`, nous intégrons l'image bitmap en deuxième page du PDF :

```
myPDF.addPage();

var donneesBitmap:Logo = new Logo(0,0);

var imageLogo:Bitmap = new Bitmap ( donneesBitmap );

myPDF.addImage ( imageLogo );
```

La figure 20-21 illustre le résultat :



*Figure 20-21. Image intégrée au PDF.*

Ainsi se termine notre aventure binaire. Au cours du prochain chapitre, nous mettrons en application la plupart des concepts abordés durant l'ensemble de l'ouvrage.

# 21

## Application finale

<b>DECOMPOSER L'APPLICATION .....</b>	<b>1</b>
<b>GERER LES CHEMINS D'ACCES .....</b>	<b>3</b>
CREATION D'UN FICHIER XML DE CONFIGURATION .....	3
<b>UTILISER DES LIBRAIRIES PARTAGEES .....</b>	<b>6</b>
GENERER ET CHARGER LES LIBRAIRIES .....	8
UTILISER LES LIBRAIRIES PARTAGEES .....	15

### Décomposer l'application

Au cours de ce chapitre, nous allons nous intéresser aux différents points essentiels à chaque application ActionScript 3. Afin de permettre à chacun d'apprécier ce chapitre, nous ne traiterons pas de cas particulier mais nous nous attarderons sur différents concepts réutilisables pour chaque type d'application.

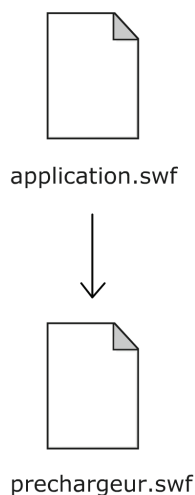
Le premier point que nous allons aborder concerne la décomposition de notre application.

La première approche consiste à ce que l'application se précharge elle-même, cette technique s'avère limitée car nous devons nous assurer qu'un minimum d'éléments soient exporté sur la première image de notre application, au risque de voir notre barre de préchargement s'afficher à partir de 50%.

Nous allons donc opter pour une deuxième technique en utilisant un premier SWF qui se chargera de précharger notre application

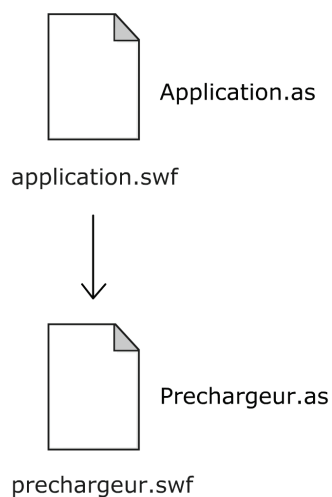
En utilisant cette approche, nous devons nous assurer que l'application préchargeant notre application soit la plus légère possible afin de ne pas nécessiter un préchargement elle aussi.

La figure 21-1 illustre le principe :



*Figure 21-1. Chargement de l'application.*

Chaque SWF possède une classe de document associée la figure 21-2 illustre l'idée :



*Figure 21-2. Classe de document.*

Nous allons définir une première classe de document nommée **Prechargeur** pour le SWF de préchargement. Celle-ci va se charger de charger tous les éléments nécessaires à notre application.

---

## A retenir

---



- Il est préférable d'utiliser un SWF dédié au préchargement de notre application.
- Ce SWF se charge de charger tous les éléments nécessaires à notre application.

## Gérer les chemins d'accès

Dans la plupart des applications ActionScript, un problème se pose éternellement lié aux chemins d'accès aux fichiers. Afin de simplifier cela nous allons définir au sein d'un fichier XML de configuration les chemins d'accès.

La première chose que fera le SWF de préchargement sera de charger ce fichier XML de configuration afin de récupérer les chemins d'accès.

## Création d'un fichier XML de configuration

Au sein d'un répertoire `init` nous créons un fichier XML nommé `initialisation.xml` contenant l'arbre XML suivant :

```
<CONFIG>
  <CHEMIN_XML chemin="xml/" />
  <CHEMIN_IMAGES chemin="images/" />
  <CHEMIN_SWF chemin="swf/" />
  <CHEMIN_POLICES chemin="polices/" />
</CONFIG>
```

Chaque nœud définit un chemin associé à un type de médias. Nous spécifions son emplacement par *FlashVars* au sein de la page HTML :

```
<script language="javascript">
  if (AC_FL_RunContent == 0) {
    alert("Cette page nécessite le fichier AC_RunActiveContent.js.");
  } else {
    AC_FL_RunContent(
      'codebase',
      'http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0,0,0',
      'width', '550',
      'height', '400',
      'src', 'chap-21-prechargeur',
      'flashvars', 'initChemin=init/',
      'quality', 'high',
      'pluginspage', 'http://www.macromedia.com/go/getflashplayer',
      'align', 'middle',
      'play', 'true',
      'loop', 'true',
      'scale', 'showall',
      'wmode', 'window',
      'devicefont', 'false',
      'id', 'chap-21-prechargeur',
      'bgcolor', '#ffffff',
      'name', 'chap-21-prechargeur',
```

```
        'menu', 'true',  
        'allowFullScreen', 'false',  
        'allowScriptAccess','sameDomain',  
        'movie', 'chap-21-prechargeur',  
        'salign', ''  
    ); //end AC code  
}  
</script>
```

Nous récupérons ce chemin afin d'indiquer au SWF de préchargement où se situe le XML de configuration.

Nous associons la classe de document suivante à notre SWF de préchargement, nous réutilisons la classe `ChargeurXML` créée au cours du chapitre 14 intitulé *Charger et envoyer des données* :

```
package org.bytearray.document  
{  
  
    import flash.events.Event;  
    import flash.events.IOErrorEvent;  
    import flash.events.SecurityErrorEvent;  
    import flash.net.URLRequest;  
    import org.bytearray.abstrait.ApplicationDefault;  
    import org.bytearray.xml.ChargeurXML;  
  
    public class Prechargeur extends ApplicationDefault  
    {  
  
        // adresse du fichier de configuration  
        private const INITIALISATION:String = "initialisation.xml";  
  
        // chemins  
        private var cheminXML:String;  
        private var cheminImages:String;  
        private var cheminSWF:String;  
        private var cheminPolices:String;  
  
        // chargement  
        private var chargeurInitialisation:ChargeurXML;  
  
        public function Prechargeur ()  
        {  
  
            chargeurInitialisation = new ChargeurXML ();  
  
            chargeurInitialisation.addEventListener ( Event.COMPLETE,  
chargeementInitTermine );  
            chargeurInitialisation.addEventListener ( IOErrorEvent.IO_ERROR,  
erreurChargement );  
            chargeurInitialisation.addEventListener (  
SecurityErrorEvent.SECURITY_ERROR, erreurChargement );  
  
            var chemin:String = loaderInfo.parameters.initChemin;  
  
            if ( chemin == null ) chemin = "init/";  
  
        }  
    }  
}
```

```
        chargeurInitialisation.charge ( new URLRequest ( chemin +  
INITIALISATION ) );  
  
    }  
  
    private function chargementInitTermine ( pEvt:Event ):void  
  
    {  
  
        var initXML:XML = pEvt.target.donnees;  
  
        cheminXML = initXML.CHEMIN_XML.@chemin;  
        cheminImages = initXML.CHEMIN_IMAGES.@chemin;  
        cheminSWF = initXML.CHEMIN_SWF.@chemin;  
        cheminPolices = initXML.CHEMIN_POLICES.@chemin;  
  
        // affiche : xml/ images/ swf/ polices/  
        trace( cheminXML, cheminImages, cheminSWF, cheminPolices );  
  
    }  
  
    private function erreurChargement ( pEvt:Event ):void  
  
    {  
  
        trace ( pEvt );  
  
    }  
  
    }  
  
}
```

Le fichier de configuration est automatiquement chargé depuis son dossier. Grâce à la variable `initChemin`, nous pouvons passer dynamiquement le chemin d'accès au fichier de configuration.

Afin de pouvoir tester notre application au sein de l'environnement auteur, nous avons ajouté le test suivant :

```
if ( chemin == null ) chemin = "init/";
```

Ce test permet simplement de définir la variable `chemin` lorsque nous testons l'application lors du développement au sein de l'environnement auteur et que l'utilisation des *FlashVars* est impossible.

Ainsi, il nous est facile de spécifier l'emplacement de ce fichier de configuration. Si lors du déploiement de notre application, le chargement d'éléments échoue, nous pouvons modifier les chemins grâce au fichier de configuration XML.

---

## A retenir

---

- L'utilisation d'un fichier XML de configuration est recommandée afin de pouvoir facilement spécifier les chemins d'accès aux fichiers.
- Le chemin d'accès au fichier de configuration XML est spécifié par *FlashVars*.
- Une fois l'application mise en production, la modification des chemins d'accès aux fichiers est simplifiée.

## Utiliser des librairies partagées

Lors du développement d'une application ActionScript 3, celle-ci est généralement divisée en plusieurs modules. Nous devons donc prendre en considération plusieurs points.

Le premier concerne l'optimisation du poids de l'application globale. Afin d'éviter de dupliquer les éléments graphiques et classes au sein des différents SWF nous allons utiliser le concept de librairie partagée abordée au cours du chapitre 13 intitulé *Chargement de contenu*.

Souvenez-vous, en plaçant les définitions de classes au sein d'un SWF nous avons extrait dynamiquement la définition associée et utilisée la classe au sein de notre application. Nous allons utiliser ce mécanisme de librairie partagée pour les éléments graphiques ainsi que les polices utilisées.

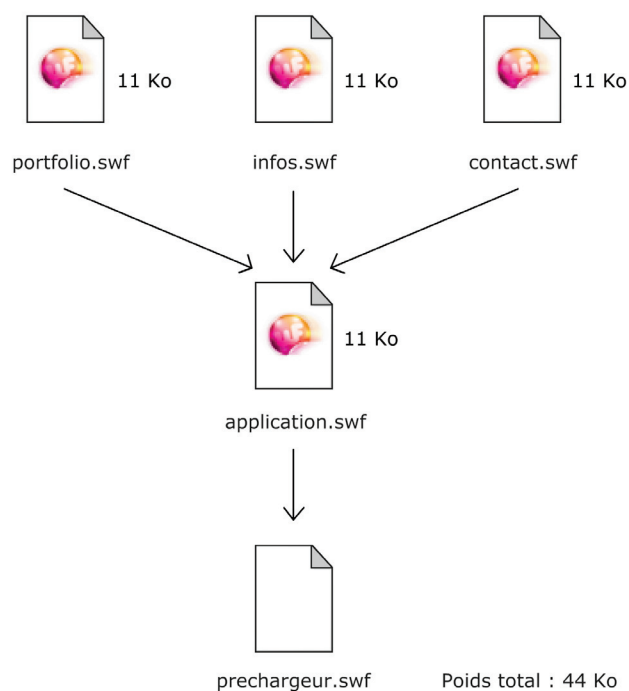
Les éléments graphiques seront chargés par le SWF de préchargement et rendues disponibles à tous les SWF constituant l'application. Nous économiserons ainsi du poids et gagnerons en facilité de mise à jour.

---

En mettant à jour la librairie partagée, toute l'application sera mise à jour sans la nécessité d'être recompilée car tout sera extrait dynamiquement.

---

Imaginons que nous souhaitons utiliser le logo d'une société au sein de l'application. Si nous n'optimisons pas celle-ci, voici comment celle-ci serait organisée :



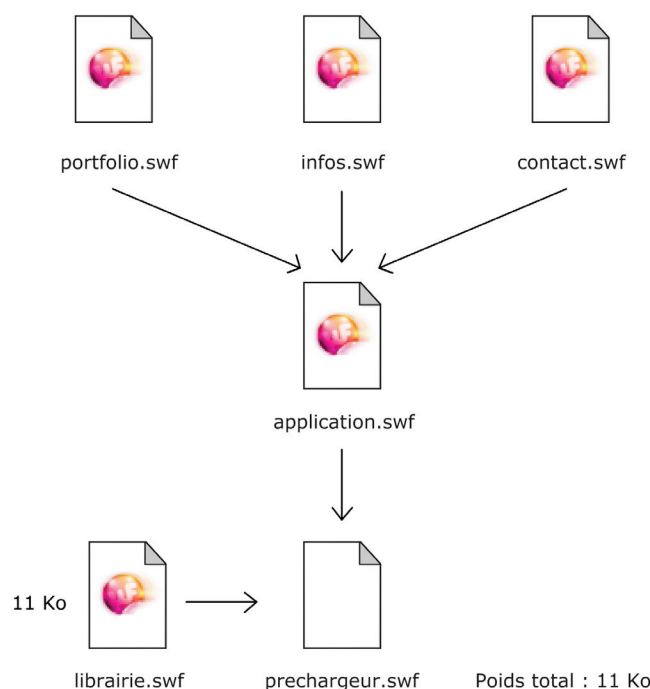
*Figure 21-3. Duplication du poids de l'image.*

Notre application est composée de nombreux SWF contenant chacun l'image, ce qui augmente sensiblement le poids total de l'application.

En utilisant une librairie partagée, le SWF de préchargement va extraire les classes nécessaires et permettre aux différents SWF constituant l'application leur utilisation.

Comme la figure 21-3 l'illustre, une image bitmap est utilisée par chacun des SWF. En dupliquant la classe `BitmapData` dans chaque SWF nous dupliquons et augmentons le poids de l'application globale ainsi que la bande passant utilisée.

Nous allons générer un SWF contenant l'image à réutiliser puis la réutiliser au sein des différents SWF, la figure 21-4 illustre le résultat :



*Figure 21-4. Chargement de l'image à l'aide d'une librairie partagée.*

Grâce à ce procédé nous optimisons le poids de l'application mais bénéficions d'un autre avantage lié à la mise à jour de notre application. En centralisant les éléments chargés et en les réutilisant, nous facilitons la mise à jour de l'application.

Nous allons nous intéresser à cette partie dans cette nouvelle partie.

## A retenir

- Afin d'optimiser la bande passante utilisée il est recommandé d'utiliser les librairies partagées.
- Leur utilisation permet de faciliter la mise à jour de l'application.
- Toutes les définitions de classes sont extraites dynamiquement, la mise à jour de l'application ne nécessite donc pas de recompilation.

## Générer et charger les librairies

Afin de générer une librairie utilisable, nous créons un nouveau document Flash CS3 et importons une image dans la bibliothèque que nous associons à une classe nommée `Logo`.

Une fois définie, nous exportons un SWF sous le nom de `libraire.swf`.

Afin d'utiliser les définitions de classes issues de la librairie, le SWF gérant le chargement doit d'abord charger la librairie partagée afin de rendre les classes disponibles par tous les SWF de l'application.

Pour cela, nous devons charger la librairie dans le domaine d'application en cours. Souvenez-vous nous avons utilisé un objet `LoaderContext` lors du chapitre 13 afin de spécifier le domaine d'application dans lequel placer les classes chargées.

Nous modifions la classe de document `Prechargeur` afin de charger en premier temps la librairie partagée, puis le SWF principal `application.swf`. Nous devons donc charger notre librairie dans un premier temps, puis charger le module `application.swf`.

Afin de charger ce contenu de manière séquentielle, nous utilisons la classe de chargement séquentielle suivante :

```
package org.bytearray.chargement

{

    import flash.display.Loader;
    import flash.display.LoaderInfo;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.ProgressEvent;
    import flash.events.SecurityErrorEvent;
    import flash.net.URLRequest;
    import flash.system.ApplicationDomain;
    import flash.system.LoaderContext;

    public class ChargeurSequentiel extends Sprite

    {

        private var liste:Array;
        private var chargeur:Loader;
        private var contexte:LoaderContext;
        private var domaine:ApplicationDomain;
        public var contentLoaderInfo:LoaderInfo;

        public function ChargeurSequentiel ( pDomaine:ApplicationDomain )

        {

            liste = new Array();

            domaine = pDomaine;

            chargeur = new Loader();

            contentLoaderInfo = chargeur.contentLoaderInfo;

            addChild ( chargeur );

            contexte = new LoaderContext ( false, domaine );
```

```
        chargeur.contentLoaderInfo.addEventListener ( Event.INIT,
redirigeEvenement );
        chargeur.contentLoaderInfo.addEventListener (
ProgressEvent.PROGRESS, redirigeEvenement );
        chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );
        chargeur.contentLoaderInfo.addEventListener (
IOErrorEvent.IO_ERROR, redirigeEvenement );
        chargeur.contentLoaderInfo.addEventListener (
SecurityErrorEvent.SECURITY_ERROR, redirigeEvenement );

    }

    private function chargementTermine ( pEvt:Event ):void
    {

        suivant();

    }

    private function redirigeEvenement ( pEvt:Event ):void
    {

        dispatchEvent ( pEvt );

    }

    public function ajoute ( pFichier:String ):void
    {

        liste.push ( new URLRequest ( pFichier ) );

    }

    public function démarre ( ):void
    {

        suivant();

    }

    private function suivant ( ):void
    {

        if ( liste.length ) chargeur.load ( liste.shift(), contexte );

        else dispatchEvent ( new Event ( Event.COMPLETE ) );

    }

}

}
```



Cette classe possède une méthode `ajoute` qui place chaque média en file d'attente, puis nous lançons le chargement des éléments à l'aide de la méthode `demarre`.

Nous mettons à jour le code de la classe `Prechargeur` afin de charger la librairie partagée :

```
package org.bytearray.document

{

    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.ProgressEvent;
    import flash.events.SecurityErrorEvent;
    import flash.net.URLRequest;
    import flash.system.ApplicationDomain;
    import org.bytearray.abstrait.ApplicationDefault;
    import org.bytearray.xml.ChargeurXML;
    import org.bytearray.chargement.ChargeurSequentiel;

    public class Prechargeur extends ApplicationDefault

    {

        private const INITIALISATION:String = "initialisation.xml";

        // chemins
        private var cheminXML:String;
        private var cheminImages:String;
        private var cheminSWF:String;
        private var cheminPolices:String;

        // chargement
        private var chargeurInitialisation:ChargeurXML;
        private var chargeur:ChargeurSequentiel;

        public function Prechargeur ()

        {

            chargeur = new ChargeurSequentiel (
ApplicationDomain.currentDomain );

            chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
            chargeur.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
            chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );
            chargeur.addEventListener ( SecurityErrorEvent.SECURITY_ERROR,
erreurChargement );

            addChild ( chargeur );

            chargeurInitialisation = new ChargeurXML ();

            chargeurInitialisation.addEventListener ( Event.COMPLETE,
chargementInitTermine );
            chargeurInitialisation.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );
```

```
        chargeurInitialisation.addEventListener (
SecurityErrorEvent.SECURITY_ERROR, erreurChargement );

        var chemin:String = loaderInfo.parameters.initChemin;

        if ( chemin == null ) chemin = "init/";

        chargeurInitialisation.charge ( new URLRequest ( chemin +
INITIALISATION ) );

    }

    private function chargementTermine ( pEvt:Event ):void
    {

        trace("chargement des libs et swf terminé !");

    }

    private function chargementEnCours ( pEvt:ProgressEvent ):void
    {

        trace("chargement en cours");

    }

    private function chargementInitTermine ( pEvt:Event ):void
    {

        var initXML:XML = pEvt.target.donnees;

        cheminXML = initXML.CHEMIN_XML.@chemin;
        cheminImages = initXML.CHEMIN_IMAGES.@chemin;
        cheminSWF = initXML.CHEMIN_SWF.@chemin;
        cheminPolices = initXML.CHEMIN_POLICES.@chemin;

        chargeur.ajoute ( cheminSWF + "librairie.swf" );

        chargeur.demarre();

    }

    private function erreurChargement ( pEvt:Event ):void
    {

        trace ( pEvt );

    }

}

}
```

Nous passons au constructeur de la classe `ChargeurSequentiel` le domaine d'application en cours. Ainsi les classes issues des SWF chargés dynamiquement sont placées au sein du domaine d'application en cours.

Afin d'indiquer l'état de chargement, nous ajoutons une barre de préchargement :

```
package org.bytearray.document

{

    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.ProgressEvent;
    import flash.events.SecurityErrorEvent;
    import flash.net.URLRequest;
    import flash.system.ApplicationDomain;
    import org.bytearray.abstrait.ApplicationDefault;
    import org.bytearray.evenements.EvenementInfos;
    import org.bytearray.xml.ChargeurXML;
    import org.bytearray.chargement.ChargeurSequentiel;

    public class Prechargeur extends ApplicationDefault

    {

        private const INITIALISATION:String = "initialisation.xml";

        // chemins
        private var cheminXML:String;
        private var cheminImages:String;
        private var cheminSWF:String;
        private var cheminPolices:String;

        // chargement
        private var chargeurInitialisation:ChargeurXML;
        private var chargeur:ChargeurSequentiel;

        // barre de préchargement
        private var jauge:Jauge;

        public function Prechargeur ()

        {

            jauge = new Jauge();

            jauge.x = int((stage.stageWidth - jauge.width) / 2);
            jauge.y = int((stage.stageHeight - jauge.height) / 2);

            chargeur = new ChargeurSequentiel (
ApplicationDomain.currentDomain );

            chargeur.addEventListener ( Event.COMPLETE, chargementTermine );
            chargeur.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
            chargeur.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );
            chargeur.addEventListener ( SecurityErrorEvent.SECURITY_ERROR,
erreurChargement );

            addChild ( chargeur );

            chargeurInitialisation = new ChargeurXML ();
```

```
        chargeurInitialisation.addEventListener ( Event.COMPLETE,
chargementInitTermine );
        chargeurInitialisation.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );
        chargeurInitialisation.addEventListener (
SecurityErrorEvent.SECURITY_ERROR, erreurChargement );

        var chemin:String = loaderInfo.parameters.initChemin;

        if ( chemin == null ) chemin = "init/";

        chargeurInitialisation.charge ( new URLRequest ( chemin +
INITIALISATION ) );

    }

    private function chargementTermine ( pEvt:Event ):void
    {

        stage.removeChild ( jauge );

    }

    private function chargementEnCours ( pEvt:ProgressEvent ):void
    {

        jauge.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;

    }

    private function chargementInitTermine ( pEvt:Event ):void
    {

        var initXML:XML = pEvt.target.donnees;

        cheminXML = initXML.CHEMIN_XML.@chemin;
        cheminImages = initXML.CHEMIN_IMAGES.@chemin;
        cheminSWF = initXML.CHEMIN_SWF.@chemin;
        cheminPolices = initXML.CHEMIN_POLICES.@chemin;

        chargeur.ajoute ( cheminSWF + "librairie.swf" );

        chargeur.demarre();

        stage.addChild ( jauge );

    }

    private function erreurChargement ( pEvt:Event ):void
    {

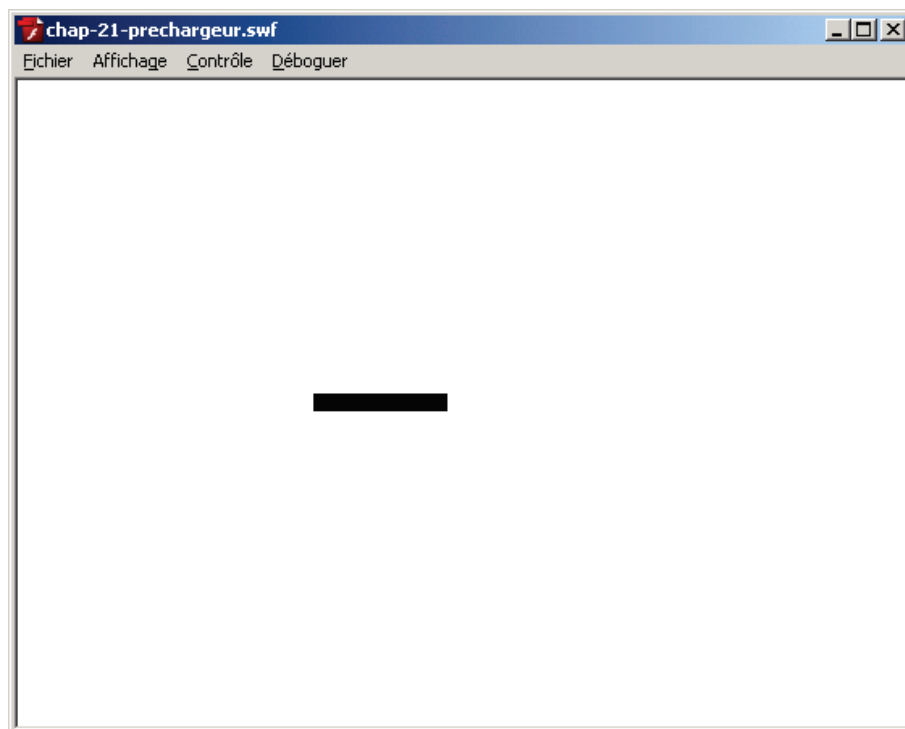
        trace ( pEvt );

    }

}

}
```

Le SWF de préchargement charge dans un premier temps la librairie contenant les définitions de classes :



*Figure 21-5. Préchargement des définitions de classes.*

Une fois les définitions chargées, celles-ci pourront être utilisées par les SWF constituant l'application.

## A retenir

- En chargeant la librairie partagée dans le domaine d'application en cours. Chaque SWF constituant l'application peut extraire les définitions de classe nécessaires.

## Utiliser les librairies partagées

Au sein du SWF `application.swf`, nous associons la classe de document suivante :

```
package org.bytearray.document
{
    import flash.display.BitmapData;
    import flash.display.Bitmap;
    import flash.system.ApplicationDomain;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Application extends ApplicationDefault
    {
```

```
private var logo:BitmapData;
private var domaineEnCours:ApplicationDomain;
private const NOM_LOGO:String = "Logo";

public function Application ()

{

    domaineEnCours = ApplicationDomain.currentDomain;

    if ( domaineEnCours.hasDefinition ( NOM_LOGO ) )

    {

        var defLogo:Class = Class ( domaineEnCours.getDefinition (
NOM_LOGO ) );

        logo = new defLogo (0,0);

        addChild ( new Bitmap ( logo ) );

    }

}

}
```

Nous extrayons dynamiquement la définition de classe `Logo` à l'aide de la méthode `getDefinition` puis nous l'instancions.

Puis le SWF `application.swf` est chargé en modifiant la classe de document du SWF de préchargement :

```
private function chargementInitTermine ( pEvt:Event ):void

{

    var initXML:XML = pEvt.target.donnees;

    cheminXML = initXML.CHEMIN_XML.@chemin;
    cheminImages = initXML.CHEMIN_IMAGES.@chemin;
    cheminSWF = initXML.CHEMIN_SWF.@chemin;
    cheminPolices = initXML.CHEMIN_POLICES.@chemin;

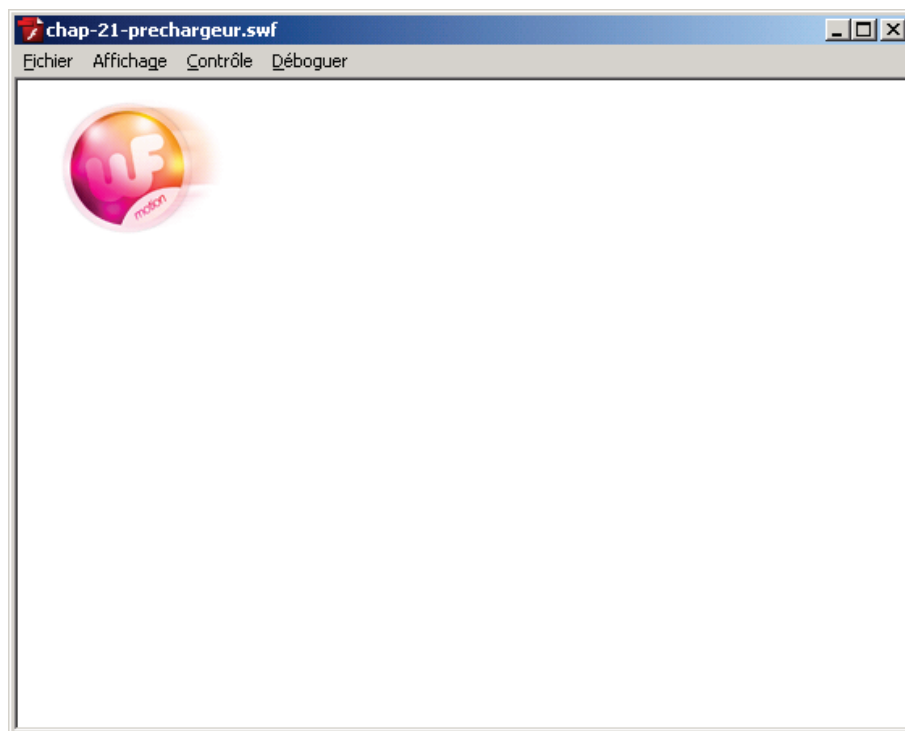
    chargeur.ajoute ( cheminSWF + "librairie.swf" );
    chargeur.ajoute ( cheminSWF + "application.swf" );

    chargeur.demarre();

    stage.addChild ( jauge );

}
```

En testant notre application, le SWF de préchargement charge la librairie partagée puis charge le SWF `application.swf` comme l'illustre la figure 21-6 :

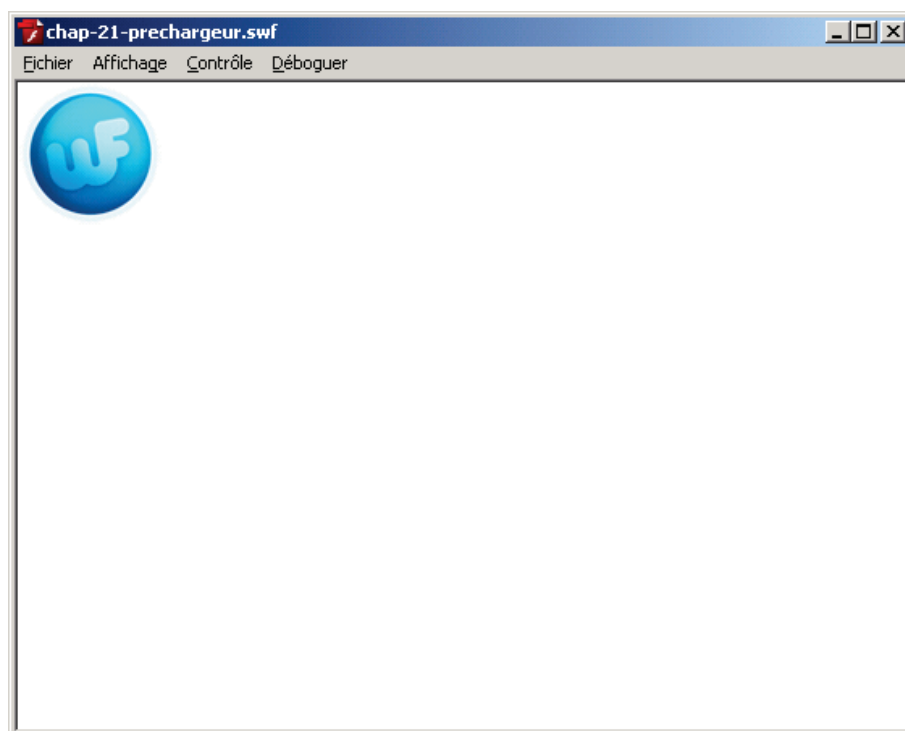


*Figure 21-6. Utilisation de la définition de classe*

*Logo.*

Afin de nous rendre compte de la facilité de mise à jour de notre application nous ouvrons le fichier `librairie.fl` et modifions l'image `Logo` dans la bibliothèque.

Nous exportons à nouveau la librairie partagée puis nous rechargeons notre site sans aucune recompilation, la figure 21-7 illustre le résultat :



*Figure 21-7. Remplacement de l'image à travers une librairie partagée.*

Sans recompiler notre application nous avons mis à jour le logo utilisé au sein du module `application.swf`.

Bien que cette pratique ait un avantage certain en matière de poids et de mise à jour, celle-ci possède néanmoins un désavantage. Nous sommes obligés de passer par le SWF de préchargement afin de tester notre module SWF car les définitions de classes utilisées par ce dernier sont chargées par le module de préchargement principal.

En testant notre application nous remarquons que la définition de classe `Logo` est correctement extraire et utilisée au sein du SWF `application.swf`.

Nous allons à présent développer un menu au sein de notre module `application.swf`. Ce menu aura pour but de charger les autres modules `portfolio.swf`, `infos.swf` et `contact.swf`.

Nous allons utiliser la même technique afin de charger dynamiquement les polices de notre menu. Pour cela, nous intégrons au sein de la librairie une police au sein de la bibliothèque.

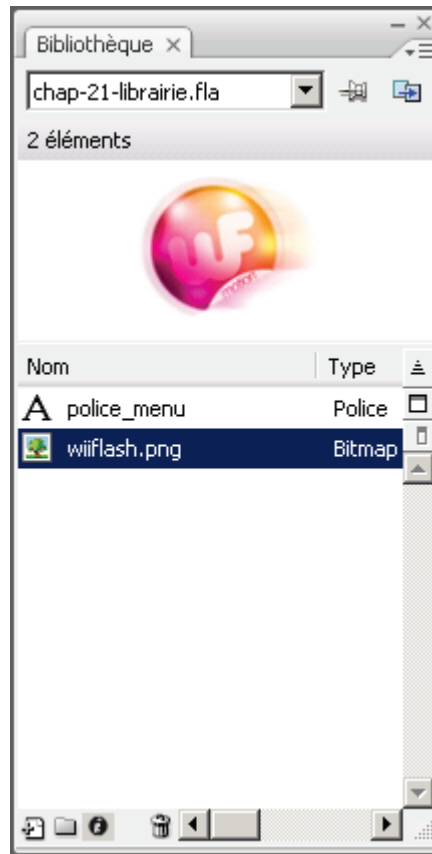
Nous obtenons donc deux éléments dans notre librairie :

- Une image bitmap représentant notre logo



- Une police utilisée par notre menu

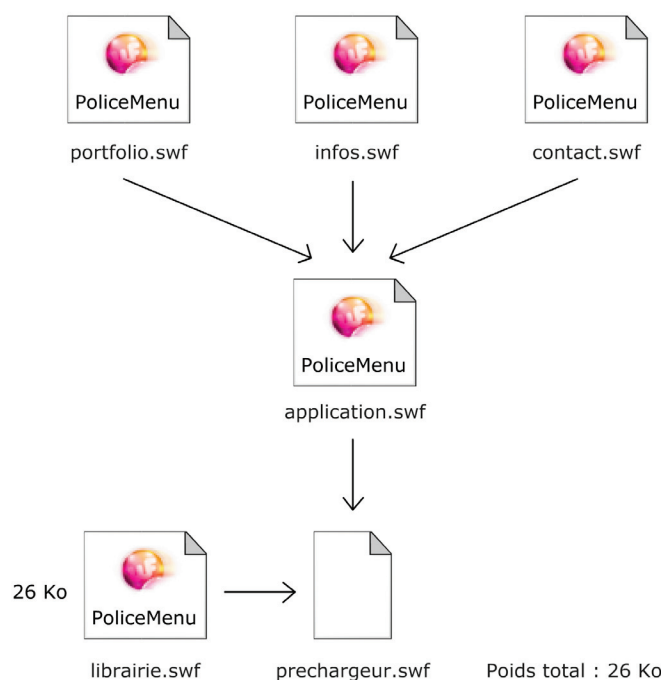
La figure 21-8 illustre la bibliothèque :



*Figure 21-8. Bibliothèque de la librairie partagée.*

Comme nous l'avons vu lors du chapitre 16 intitulé *Le texte*, les polices peuvent être embarquées à l'exécution. De la même manière que notre image bitmap, nous allons extraire la définition de classe de police et ajouter celle-ci dans la liste des polices disponibles.

La figure 21-9 illustre l'idée :



*Figure 21-9. Chargement d'une police à l'aide d'une librairie partagée.*

Nous modifions la classe `Application` afin d'extraire la police et l'ajouter :

```

package org.bytearray.document
{
    import flash.display.BitmapData;
    import flash.display.Bitmap;
    import flash.system.ApplicationDomain;
    import flash.text.Font;
    import flash.text.TextField;
    import flash.text.TextFormat;
    import org.bytearray.abstrait.ApplicationDefault;

    public class Application extends ApplicationDefault
    {
        private var logo:BitmapData;
        private var police:Font;
        private const NOM_LOGO:String = "Logo";
        private const NOM_POLICE:String = "PoliceMenu";

        public function Application ()
        {

```

```
        if ( ApplicationDomain.currentDomain.hasDefinition ( NOM_LOGO ) )
        {
            var defLogo:Class = Class (
ApplicationDomain.currentDomain.getDefinition ( NOM_LOGO ) );

            logo = new defLogo (0,0);

            addChild ( new Bitmap ( logo ) );

        }

        if ( ApplicationDomain.currentDomain.hasDefinition ( NOM_POLICE
))

        {

            var defPolice:Class = Class (
ApplicationDomain.currentDomain.getDefinition ( NOM_POLICE ) );

            police = new defPolice();

            Font.registerFont ( defPolice );

        }

    }

}
```

Afin d'utiliser la police extraite, nous créons un menu dynamique. Nous avons donc besoin de charger un fichier XML contenant les chemins d'accès au XML. Pour cela, nous allons diffuser un événement depuis le SWF de préchargement, cela va nous permettre de passer au module `application.swf` les chemins d'accès aux fichiers.

Nous définissons un fichier XML nommé `donnees.xml` :

```
<MENU>
<BOUTON legende="Accueil" url="accueil.swf"/>
<BOUTON legende="Photos" url="photos.swf"/>
<BOUTON legende="Blog" url="blog.swf"/>
</MENU>
```

Puis nous créons une classe événementielle `EvenementInfos` contenant les différentes propriétés indiquant les chemins :

```
package org.bytearray.events

{
    import flash.events.Event;

    public class EvenementInfos extends Event

    {
```

```
        public static const CHEMINS:String = "chemins";

        public var images:String;
        public var polices:String;
        public var swf:String;
        public var xml:String;

        public function EvenementInfos ( pType:String, pCheminImages:String,
pCheminPolices:String, pCheminSWF:String, pCheminXML:String )

        {

            super ( pType );

            images = pCheminImages;
            polices = pCheminPolices;
            swf = pCheminSWF;
            xml = pCheminXML;

        }

    }
}
```

Le SWF de préchargement doit donc indiquer au SWF principal les chemins d'accès aux fichiers, pour cela nous diffusons un événement au module `application.swf` une fois les fichiers chargés :

```
private function chargementTermine ( pEvt:Event ):void

{

    stage.removeChild ( jauge );

    pEvt.target.contentLoaderInfo.sharedEvents.dispatchEvent ( new
EvenementInfos ( EvenementInfos.CHEMINS , cheminImages, cheminPolices,
cheminSWF, cheminXML ) );

}
```

Notre SWF `application.swf` n'a plus qu'à écouter cet événement et enregistrer les chemins puis charger le XML afin de créer le menu :

```
package org.bytearray.document

{

    import flash.display.BitmapData;
    import flash.display.Bitmap;
    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.IOErrorEvent;
    import flash.events.MouseEvent;
    import flash.events.SecurityErrorEvent;
    import flash.net.URLRequest;
    import flash.system.ApplicationDomain;
    import flash.text.Font;
    import flash.text.TextFormat;
    import org.bytearray.evenements.EvenementInfos;
```

```
import org.bytearray.xml.ChargeurXML;
import org.bytearray.abstrait.ApplicationDefaut;

public class Application extends ApplicationDefaut
{
    private var logo:BitmapData;
    private var police:Font;
    private var donneesXML:ChargeurXML;
    private var domaineEnCours:ApplicationDomain;
    private var conteneurMenu:Sprite;
    private var chargeur:Loader;

    private const NOM_LOGO:String = "Logo";
    private const NOM_POLICE:String = "PoliceMenu";
    private const MENU_XML:String = "donnees.xml";

    private var cheminXML:String;
    private var cheminImages:String;
    private var cheminSWF:String;
    private var cheminPolices:String;

    public function Application ()
    {
        loaderInfo.sharedEvents.addEventListener( EvenementInfos.CHEMINS,
initialisation );

        chargeur = new Loader();

        addChild ( chargeur );

        conteneurMenu = new Sprite();

        conteneurMenu.addEventListener ( MouseEvent.CLICK, clicBouton,
true );

        conteneurMenu.x = 110;
        conteneurMenu.y = 25;

        addChildAt ( conteneurMenu, 0 );

        donneesXML = new ChargeurXML ( true );

        donneesXML.addEventListener ( Event.COMPLETE, chargementTermine
);
        donneesXML.addEventListener ( IOErrorEvent.IO_ERROR,
erreurChargement );
        donneesXML.addEventListener ( SecurityErrorEvent.SECURITY_ERROR,
erreurChargement );

        domaineEnCours = ApplicationDomain.currentDomain;

        if ( domaineEnCours.hasDefinition ( NOM_LOGO ) )
        {
            var defLogo:Class = Class ( domaineEnCours.getDefinition (
NOM_LOGO ) );
```

```

        logo = new defLogo (0,0);

        addChild ( new Bitmap ( logo ) );

    }

}

private function initialisation ( pEvt:EvenementInfos ):void
{
    cheminXML = pEvt.xml;
    cheminImages = pEvt.images;
    cheminSWF = pEvt.swf;
    cheminPolices = pEvt.polices;

    if ( domaineEnCours.hasDefinition ( NOM_POLICE ))
    {
        var defPolice:Class = Class ( domaineEnCours.getDefinition (
NOM_POLICE ) );

        police = new defPolice();

        Font.registerFont ( defPolice );

        donneesXML.charge ( new URLRequest ( cheminXML + MENU_XML )
);

    }

}

private function chargementTermine ( pEvt:Event ):void
{
    var donnees:XML = pEvt.target.donnees;

    var boutonMenu:Bouton;
    var i:int = 0;

    var formatage:TextFormat = new TextFormat ( police.fontName, 8 );

    for each ( var noeud:XML in donnees.* )
    {
        boutonMenu = new Bouton();

        boutonMenu.buttonMode = true;
        boutonMenu.mouseChildren = false;
        boutonMenu.lien = noeud.@url;

        boutonMenu.legende.embedFonts = true;

        boutonMenu.legende.defaultTextFormat = formatage;

        boutonMenu.legende.text = noeud.@legende;
    }
}

```

```
        boutonMenu.x = (boutonMenu.width + 5) * i;

        conteneurMenu.addChild ( boutonMenu );

        i++;

    }

}

private function clicBouton ( pEvt:MouseEvent ):void
{
    chargeur.load ( new URLRequest ( cheminSWF + pEvt.target.lien )
);
}

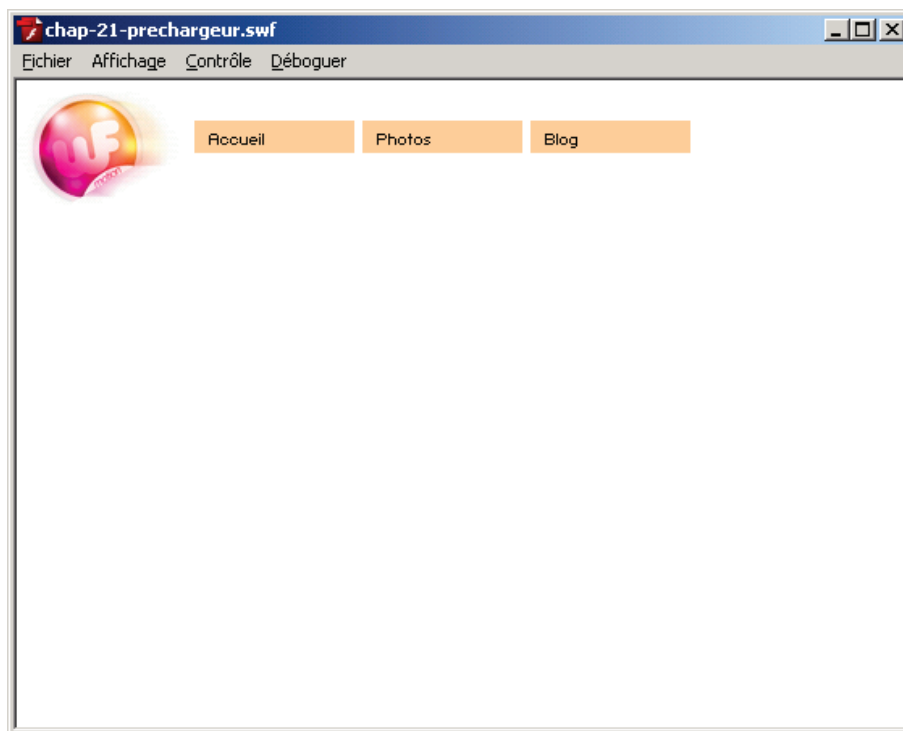
private function erreurChargement ( pEvt:Event ):void
{
    trace(pEvt);
}

}

}
```

En utilisant un nom de police générique, nous pouvons ainsi remplacer plus tard la police utilisée pour le menu par une autre sans créer des erreurs de logique. Ainsi une police Verdana, Times ou autre pourra être utilisée de manière transparente pour désigner la police du menu.

Lorsque nous testons le module `application.swf`, nous ne voyons pas le menu affiché. A l'inverse, en testant le SWF par le module principal nous obtenons le résultat suivant :



*Figure 21-10. Utilisation de la police au sein du menu dynamique.*

La police est ainsi intégrée au sein du site et chargée une seule et unique fois. Nous économisons bande passante et gagnons en facilité de mise à jour.

Afin de mettre à jour la police, nous modifions la police présente au sein de la bibliothèque et régénérons le fichier `librairie.swf`.

La figure 21-11 illustre l'application relancée en ayant modifié la police présente au sein de la librairie partagée :



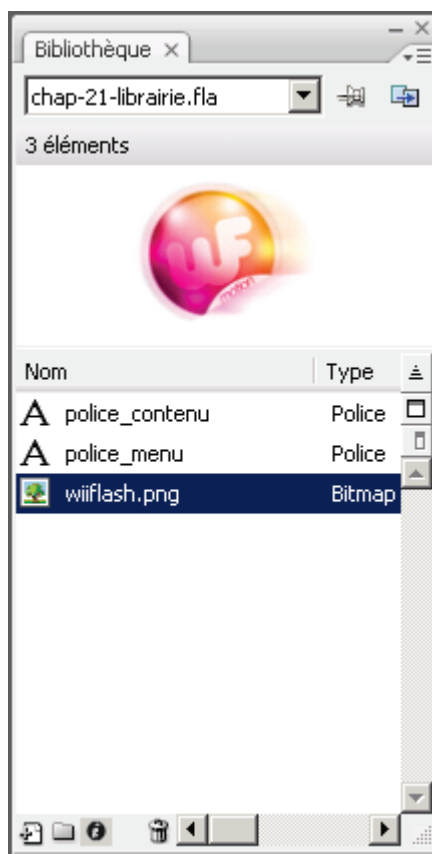


*Figure 21-11. Remplacement de la police à travers la librairie partagée.*

La police extraite est désormais disponible au sein de tous les SWF de l'application. En cliquant sur chacun des boutons nous chargeons de nouveaux SWF ayant recours à cette police.

Nous ajoutons au sein de la librairie partagée une nouvelle police destinée à être utilisée par le contenu. Celle-ci est associée à une classe `PoliceContenu`.

La figure 21-12 illustre la bibliothèque de la librairie partagée :



*Figure 21-12. Bibliothèque de la librairie partagée.*

Chaque bouton du menu charge un SWF représentant une partie du site, ici encore chaque SWF va utiliser les définitions de polices chargées le SWF de préchargement. Nous évitons ainsi d'intégrer des polices au sein de chaque SWF et optimisons le poids du site.

En cliquant sur le bouton *Accueil* nous chargeons le SWF `accueil.swf`.

Nous ajoutons au sein de ce dernier le code suivant afin de créer un champ texte et d'utiliser la définition de police `PoliceContenu` :

```
var domaineEnCours:ApplicationDomain = ApplicationDomain.currentDomain;
var policeContenu:String = "PoliceContenu";

if ( domaineEnCours.hasDefinition ( policeContenu ) )
{
    var defPolice:Class = Class ( domaineEnCours.getDefinition (
    policeContenu ) );

    var police:Font = new defPolice();

    var formatage:TextFormat = new TextFormat ( police.fontName, 8 );
```

```

Font.registerFont ( defPolice );

var monContenu:TextField = new TextField()

addChild ( monContenu );

monContenu.x = 40;
monContenu.y = 110;

monContenu.embedFonts = true;
monContenu.defaultTextFormat = formatage;
monContenu.autoSize = TextFieldAutoSize.LEFT;
monContenu.wordWrap = true;
monContenu.width = 450;
monContenu.text = "Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Duis consectetur enim fringilla ligula. Donec facilisis scelerisque sem.
Nunc lobortis ante eget turpis. Donec auctor ullamcorper diam. Phasellus sed
enim non urna aliquam consectetur. Morbi sit amet purus. Suspendisse eros leo,
volutpat eu, eleifend sit amet, bibendum ac, lacus. Suspendisse elit. In
egestas lorem in nisi. Integer imperdiet felis et ligula. Quisque semper
dapibus pede. Mauris ac neque vel erat porttitor hendrerit. Duis justo augue,
scelerisque a, rutrum nec, rutrum ut, justo. Maecenas luctus. Aenean a leo et
eros blandit sollicitudin. Sed bibendum placerat ligula. Integer varius.
Aliquam in felis in felis convallis laoreet. Vivamus nulla. Quisque rutrum
massa id nunc."

}

```

Nous extrayons dynamiquement la police `PoliceContenu`, la figure 21-13 illustre le résultat :



*Figure 21-13. Utilisation de la police au sein du texte de présentation.*

Afin de tester notre mécanisme, nous modifions le fichier de police puis nous relançons notre site sans recompiler les autres SWF constituant l'application.

Nous obtenons le résultat illustré par la figure suivante :



*Figure 21-14. Remplacement de la police à travers la librairie partagée.*

De cette manière nous centralisons nos images ainsi que les polices par le biais de la librairie partagée. La mise à jour de l'application est ainsi simplifiée et les temps de chargement optimisés.

## A retenir

- Afin d'extraire dynamiquement une classe nous utilisons la méthode `getDefinition` de l'objet `ApplicationDomain`.

Ainsi s'achève notre aventure au cœur de l'ActionScript 3. Rendez-vous sur le forum dédié à l'ouvrage pour en discuter et ainsi répondre à d'éventuelles questions.

A très vite !