3

Le modèle événementiel

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 devions écrire le code suivant :

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


Thibault Imbert
pratiqueactionscript3.bytearray.org

```
{

    // 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 :

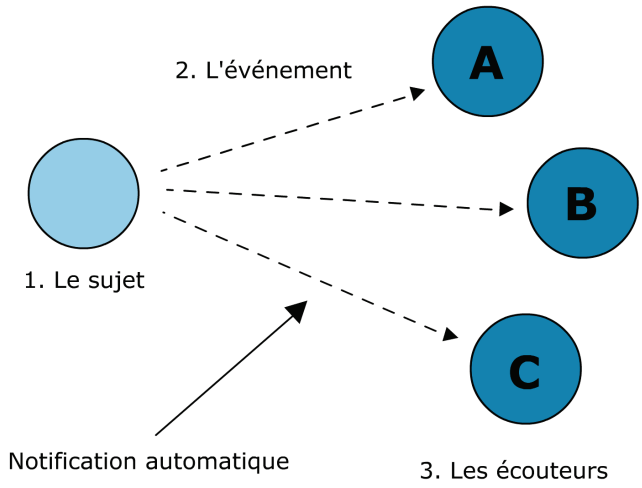

Figure 3-1. Schéma du modèle de conception Observateur

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 :

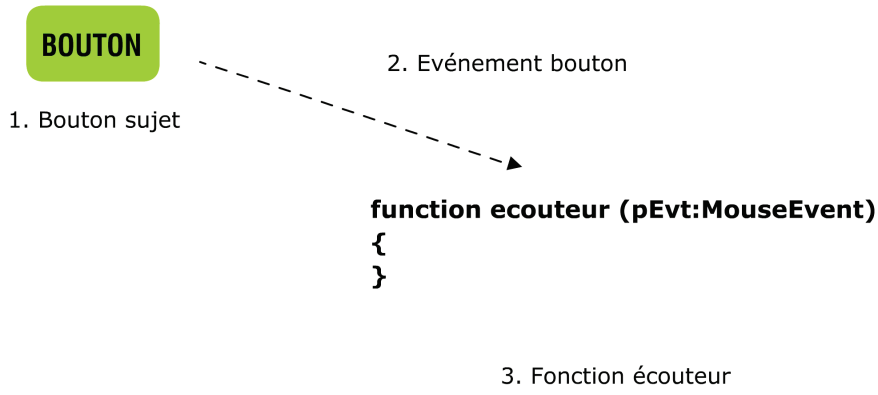

Figure 3-2. Modèle événementiel dans une application ActionScript 3

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 devions 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 devions 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
                          |
```

```
                    +-DisplayObjectContainer
                                  |
                                  +-Sprite
                                        |
                                        +-MovieClip
```

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

Ecouter 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 :

```
sujet.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 `clicck` :

```
monBouton.addEventListener( "clicck", 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.IMEEvent`
- `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 récupérons 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 récupérons 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/onMouseClick(). 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 enfonçé : false
        trace ( "ALT enfonçé : " + 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 devions 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 devrons 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 souscris. 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 *référencés* 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` :

```
// instanciation 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 dé 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 leteur Flash 9.