4

La liste d'affichage

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 :

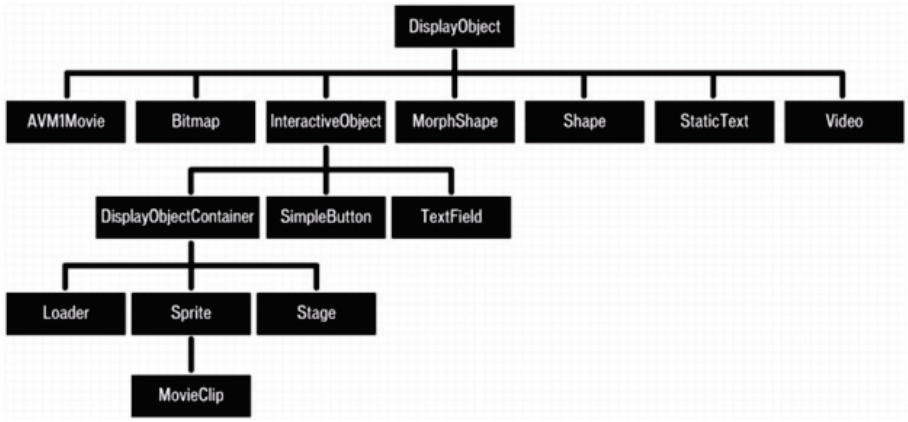

Figure 3-1. Classes graphiques de la liste d'affichage.

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 préférerons 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 :

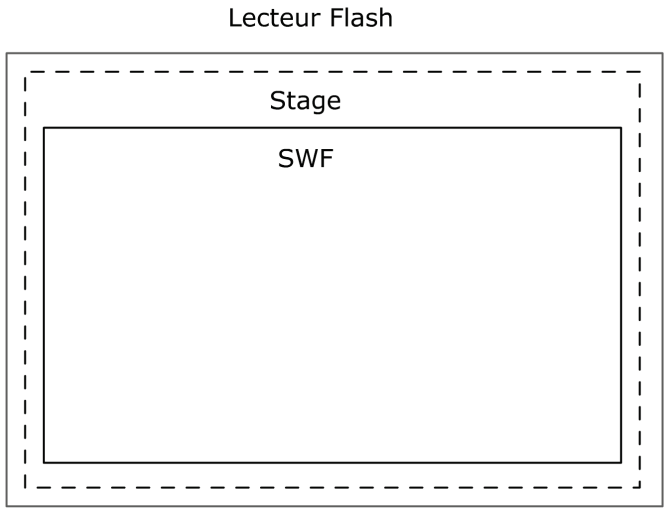

Figure 3-2. Schéma du système d'affichage du lecteur Flash.

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 :

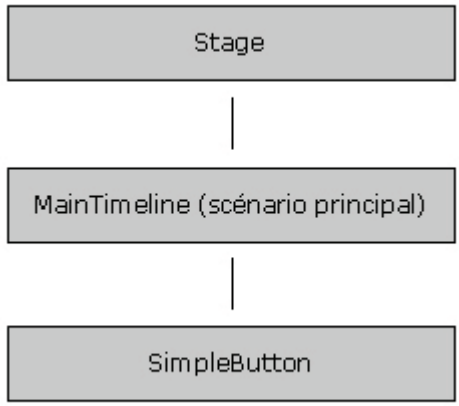

Figure 3-3. Liste d'affichage avec un bouton.

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 :

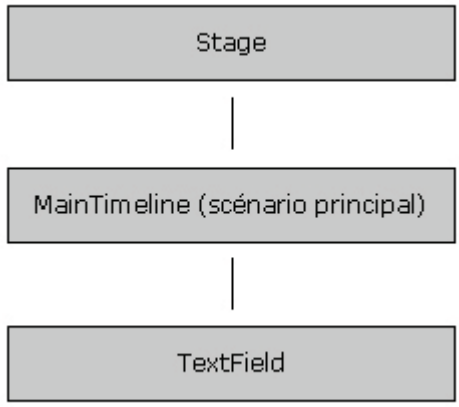

Figure 3-4. Liste d'affichage simple avec un champ texte.

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 replaçé 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 :

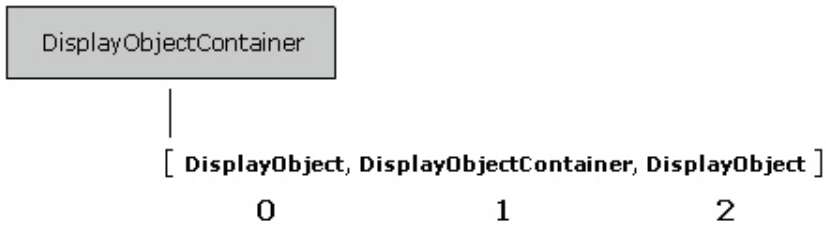

Figure 3-5. Tableau interne d'accès aux objets enfants.

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 récupérons 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 = "monOcurrence";

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 = "monOcurrence";

addChild ( monClip );

// affiche : [object MovieClip]
trace( getChildByName ("monOcurrence") );
```

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 = "monOcurrence";

// affiche : null
trace( getChildByName ("monOcurrence") );
```

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 = "monOcurrence";

addChild ( monClip );
```

```
var depart:Number = getTimer();
for ( var i:int = 0; i< 5000000; i++ )

{

    getChildByName ("monOcurrence");

}

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 = "monOcurrence";

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 avions 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 entres les objets graphiques. Ainsi les profondeurs 6, 8 et 30 pouvaient êtres 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 :

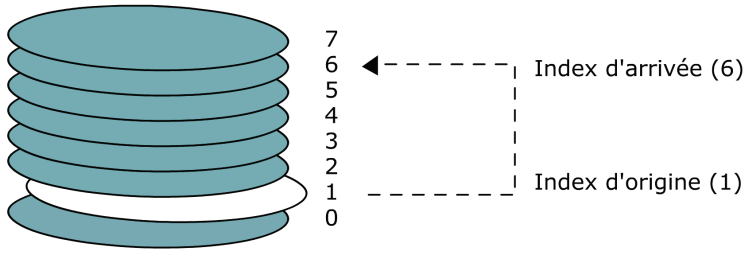

Figure 3-8. Changement d'index pour un `DisplayObject`.

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` :

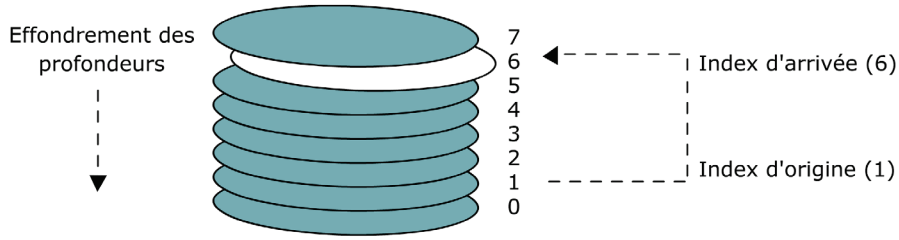

Figure 3-9. Effondrement des profondeurs.

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 :

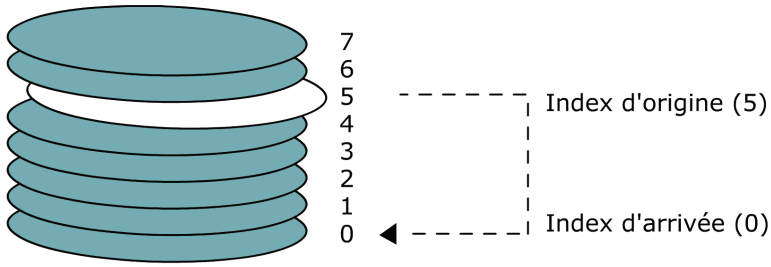

Figure 3-10. Déplacement vers le bas de la pile.

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 :

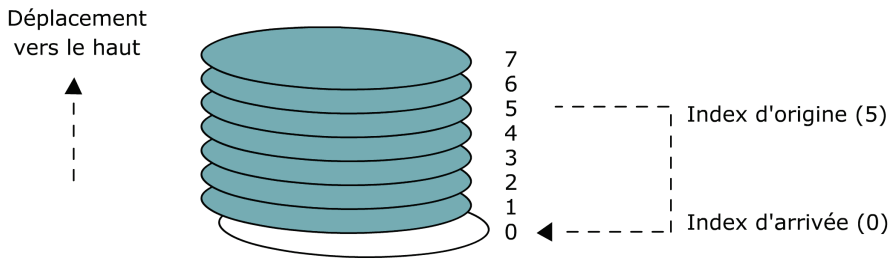

Figure 3-11. Objets graphiques poussés d'un index vers le haut de la pile.

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 avions 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` :

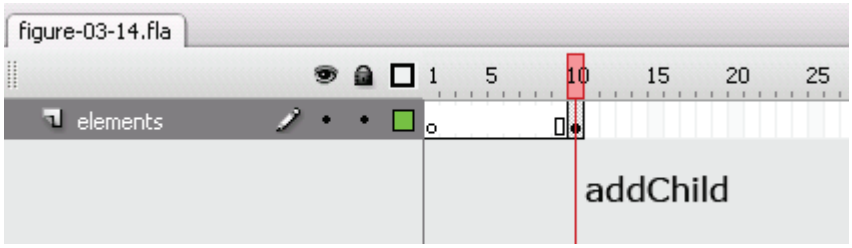

Figure 3-14. Ajout d'objets graphiques.

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` :

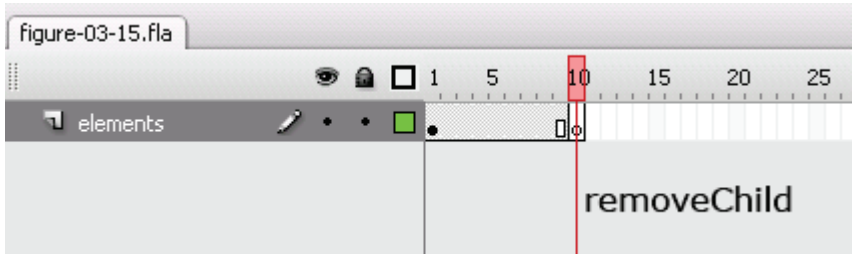

Figure 3-15. Suppression d'objets graphiques.

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.

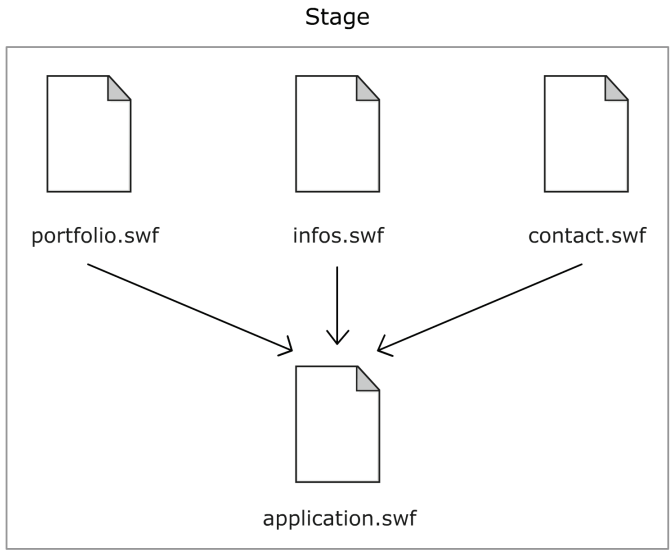

Figure 3-16. Objet `Stage`.

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 surement 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 développions 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 :

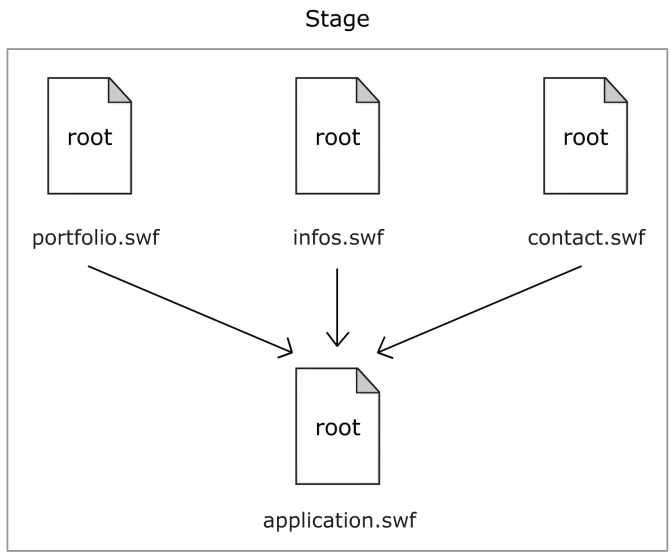

Figure 3-17. Scénarios.

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 :

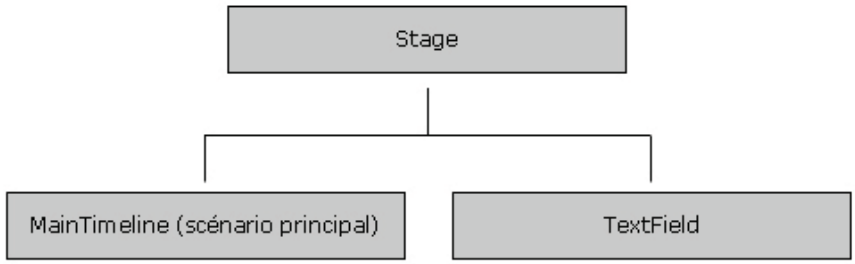

Figure 3-17. Simulation de _level.

Notre champ texte est ici un enfant direct de l'objet `Stage`, nous obtenons donc le même concept d'empilement de scénario établit 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.