2

Langage et API

Le langage ActionScript 3

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

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

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

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

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

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

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

La figure 2-1 illustre les deux entités :

Figure 2-1. Langage ActionScript 3.

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

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

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

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

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

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

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

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

```
import flash.display.MovieClip;

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

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

A retenir

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

Machines virtuelles

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

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

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

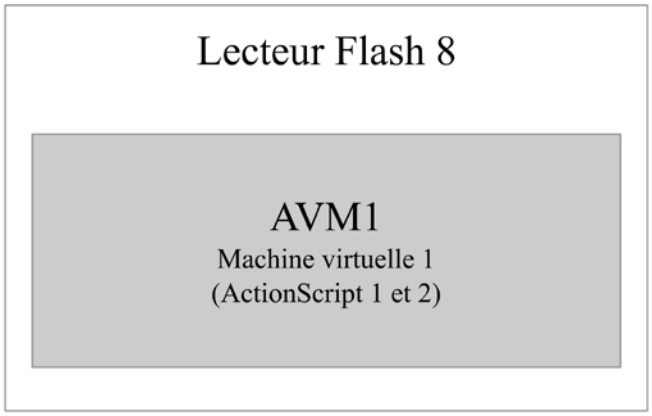

Figure 2-2. AVM1 au sein du lecteur Flash 8.

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

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

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

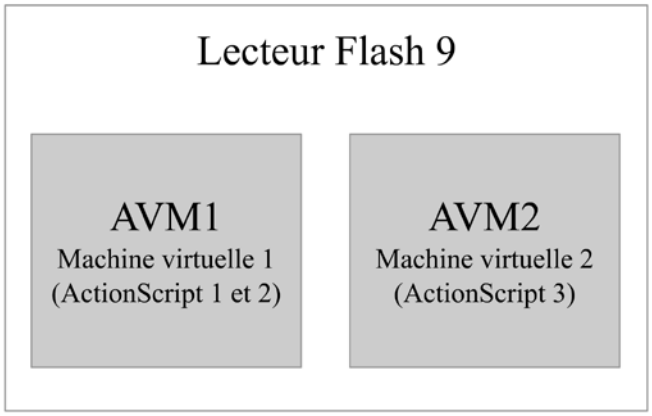

Figure 2-3. Le lecteur Flash 9 et les deux machines virtuelles AVM1 et AVM2.

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

A retenir

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

Traduction dynamique

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

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

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

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

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

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

http://en.wikipedia.org/wiki/Just-in-time_compilation

http://fr.wikipedia.org/wiki/Compilation_%C3%A0_la_vol%C3%A9e

Gestion des types à l'exécution

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

```
variable:Type
```

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

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

L'erreur suivante était générée à la compilation :

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

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

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

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

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

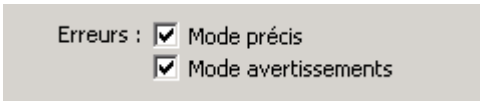

Figure 2-4. Options du compilateur ActionScript 3.

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

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

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

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

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

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

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

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

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

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

```
var distance:Number = "150.5";

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

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

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

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

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

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

```
var tableauDonnees:Array = [ "150", "250" ];

// l'entrée du tableau est automatiquement convertie en int
var distance:Number = tableauDonnees[0];
```

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

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

```
var tableauDonnees:Array = [ "clip", "250" ];

// lève une erreur à l'exécution
var clip:MovieClip = tableauDonnees[0];
```

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

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

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

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

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

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

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

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

Voici un exemple permettant de justifier cette décision :

Une simple boucle utilise une variable d'incrémentation `i` de type `int` :

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

for ( var i:int = 0; i< 500000; i++ )

{

}

// affiche : 5
trace( getTimer() - debut );
```

La boucle nécessite 5 millisecondes pour effectuer 500 000 itérations.

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

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

for ( var i = 0; i< 500000; i++ )

{

}

// affiche : 72
trace( getTimer() - debut );
```

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

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

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

for ( var i:int = 0; i< 500000; i++ )

{

    prenomRaccourci = prenom.substr ( 0, 3 );

}

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

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

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

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

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

for ( var i:int = 0; i< 500000; i++ )

{

    prenomRaccourci = prenom.substr ( 0, 3 );

}

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

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

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

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

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

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

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

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

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

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

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

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

A la compilation, l'avertissement suivant est affiché :

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

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

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

A retenir

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

Erreurs à l'exécution

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

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

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

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

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

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

La fenêtre de sortie affiche l'erreur suivante :

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

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

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

var nbImages:int;

// grâce au bloc try catch nous pouvons gérer l'erreur
try

{

    nbImages = monClip.totalFrames;

}   catch ( pErreur:Error )

{

    trace ( "une erreur d'exécution a été levée !");

}
```

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

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

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

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

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

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

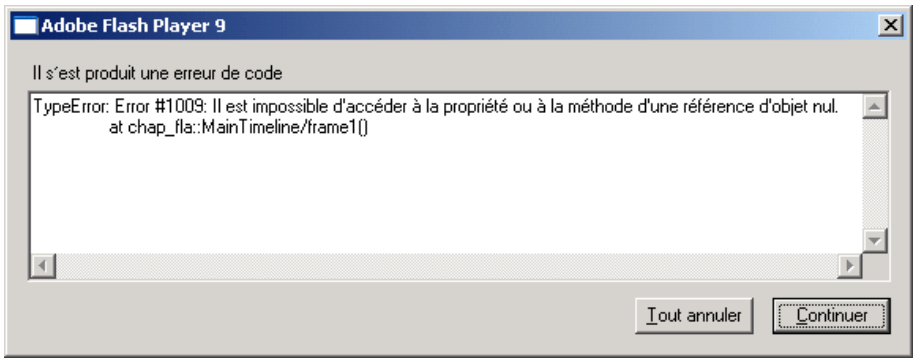

Figure 2-5. Exemple d'erreur à l'exécution.

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

A retenir

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

Nouveaux types primitifs

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

```
var age:Number = 20;

var vitesse:Number = 12.8;
```

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

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

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

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

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

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

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

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

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

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

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

```
var age:int = 22.2;

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

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

```
var age:int = 22.8;

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

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

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

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

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )

{

    arrondi = Math.floor ( distance );

}

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

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

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

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )

{

    arrondi = distance;

}

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

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

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

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

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

```
var distance:int = -3.2;

// affiche : -3
trace(distance);

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

// affiche : -4
```

```
trace( profondeur );
```

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

```
var arrondi:uint;
```

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

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

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

for ( var i:int = 0; i< 5000000; i++ )

{


}

// affiche : 61
trace( getTimer() - debut );
```

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

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

for ( var i:uint = 0; i< 5000000; i++ )

{


}

// affiche : 238
trace( getTimer() - debut );
```

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

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

for ( var i:Number = 0; i< 5000000; i++ )

{


}

// affiche : 102
trace( getTimer() - debut );
```

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

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

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

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

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )

{

    arrondi = distance + 1;

}

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

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

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

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

var debut:Number = getTimer();

for ( var i:int = 0; i< 500000; i++ )

{

    arrondi = Math.ceil ( distance ) + 1;

}

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

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

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

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

A retenir

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

Valeurs par défaut

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

```
var prenom;

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

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

```
var condition:Boolean;
var total:int;
var couleur:uint;
var resultat:Number;
var personnage:Object;
var prenom:String;
var donnees:*;

// affiche : false
trace( condition );

// affiche : 0
trace( total );

// affiche : 0
trace( couleur );

// affiche : NaN
trace( resultat );

// affiche : null
trace( personnage );

// affiche : null
trace( prenom );

// affiche : undefined
trace( donnees );
```

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

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

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

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

```
var prenom:String = "Bob";

// génère une erreur à la compilation
trace( prenom.proprieteInexistante );
```

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

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

// affiche : undefined
trace( objet.proprieteInexistante );
```

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

```
var distance:Number;

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

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

```
var distance:int;

var autreDistance:uint;

// affiche : 0
trace( distance );

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

A retenir

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

Nouveaux types composites

Deux nouveaux types composites sont intégrés en ActionScript 3 :

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

Nouveaux mots-clés

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// affiche : true
trace( MovieClip(monClip) != null );
```

Nous aurions pu transtyper avec l'écriture traditionnelle suivante :

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

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

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

Fonctions

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

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

```
function alerte ( pMessage:String ):void

{

    trace( pMessage );

}
```

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

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

Si nous omettons le paramètre :

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

L'erreur à la compilation suivante est générée :

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

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

```
function alerte ( pMessage:String="message par défaut" ):void

{

    trace( pMessage );

}
```

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

```
function alerte ( pMessage:String="message par défaut" ):void

{

    trace( pMessage );

}

// affiche : message par défaut
alerte ();
```

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

```
function alerte ( pMessage:String="message par défaut" ):void

{

    trace( pMessage );

}

// affiche : un message personnalisé !
alerte ( "un message personnalisé !" );
```

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

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

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

```
function calculMoyenne ():Number

{

    var lng:Number = arguments.length;
    var total:Number = 0;

    for (var i:Number = 0; i< lng; i++)

    {

         total += arguments[i];

    }

    return total / lng;

}

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

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

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

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

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

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

```
function calculMoyenne ( ...parametres ):Number

{

    var lng:int = parametres.length;
    var total:Number = 0;

    for (var i:Number = 0; i< lng; i++)

    {

         total += parametres[i];

    }

    return total / lng;

}

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

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

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

Contexte d'exécution

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

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

```
var personnage:Object = { age : 25, nom : "Bobby" };

// la fonction parler est passée en référence
personnage.parler = parler;

function parler ( )

{

    trace("bonjour, je m'appelle " + this.nom + ", j'ai " + this.age + " ans");

}

// appel de la méthode
// affiche : bonjour, je m'appelle Bobby, j'ai 25 ans
personnage.parler();
```

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

```
var personnage:Object = { age : 25, nom : "Bobby" };

// la fonction parler est passée en référence
personnage.parler = parler;

function parler ( )

{

    trace("bonjour, je m'appelle " + this.nom + ", j'ai " + this.age + " ans");

}

// appel de la méthode
// affiche : bonjour, je m'appelle undefined, j'ai undefined ans
personnage.parler();
```

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

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

Boucles

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

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

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

for (var p:String in personnage)

{

    /* affiche :
    age
    prenom
    */
    trace( p );

}
```

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

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

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

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

for ( var p:String in tableauDonnees )

{

    /* affiche :
    5654
    95
    54
    687968
    97851
    */
    trace( tableauDonnees[p] );

}
```

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

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

for each ( var p:* in personnage )

{

    /* affiche :
    50
    Bobby
    */
    trace( p );

}
```

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

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

for each ( var p:* in tableauDonnees )

{

    /* affiche :
    5654
    95
    54
    687968
    97851
    */
    trace( p );

}
```

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

Enrichissement de la classe Array

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

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

```
// Array.forEach procède à une navigation simple
// sur chaque élement à l'aide d'une fonction spécifique

var prenoms:Array = [ "bobby", "willy", "ritchie" ];

function navigue ( element:*, index:int, tableau:Array ):void

{

    trace ( element + " : " + index + " : " + tableau);

}
```

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

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

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

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

```
var donnees:Array = [ 12, "bobby", "willy", 58, "ritchie" ];

function navigue ( element:*, index:int, tableau:Array ):Boolean

{

    return ( element is Number );

}

var tableauNombres:Boolean = donnees.every ( navigue );

// affiche : false
trace( tableauNombres );
```

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

Un nouveau tableau de prénoms formatés est créé :

```
var prenoms:Array = ["bobby", "willy", "ritchie"];

function navigue ( element:*, index:int, tableau:Array ):String

{

    return element.charAt(0).toUpperCase()+element.substr(1).toLowerCase();

}

// on créé un tableau à partir du retour de la fonction navigue
var prenomsFormates:Array = prenoms.map ( navigue );

// affiche : Bobby,Willy,Ritchie
trace( prenomsFormates );
```

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

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

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


function navigue ( element:*, index:int, tableau:Array ):Boolean

{

    return ( element.age >= 18 );

}

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

function parcourir ( element:*, index:int, tableau:Array ):void

{

    trace ( element.prenom, element.age );

}

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

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

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


function navigue ( element:*, index:int, tableau:Array ):Boolean

{

    return ( element.sexe == "F" );

}

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

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

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

```
var utilisateurs:Array = [ "Bobby", "Linda", "Ritchie", "Stevie", "Linda" ];

var position:int = utilisateurs.indexOf ("Linda");

var positionFin:int = utilisateurs.lastIndexOf ("Linda");

// affiche : 1
trace( position );

// affiche : 4
trace( positionFin );
```

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

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

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

var monAutreClip:DisplayObject = new MovieClip();

// une référence au clip monClip est placée au sein du tableau
var tableauReferences:Array = [ monClip ];

var position:int = tableauReferences.indexOf (monClip);

var autrePosition:int = tableauReferences.lastIndexOf (monAutreClip);

// affiche : 0
trace( position );

// affiche : -1
trace( autrePosition );
```

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

A retenir

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

Ramasse-miettes

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

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

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

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

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

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

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

// génère une erreur à la compilation
delete personnage;
```

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

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

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

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

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

// affiche : Bobby
trace( personnage.prenom );

// supprime la propriété prenom
delete ( personnage.prenom );

// affiche : undefined
trace( personnage.prenom );
```

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

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

// supprime la référence vers l'objet personnage
personnage = null;
```

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

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

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

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

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

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

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

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

// suppression d'une seule référence
personnage = null;
```

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

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

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

// supprime une des deux références seulement
personnage = null;

var personnageOrigine:Object = tableauPersonnages[0];

// affiche : Bobby
trace( personnageOrigine.prenom );

// affiche : 50
trace( personnageOrigine.age );
```

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

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

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

// supprime la première référence
personnage = null;

// supprime la seconde référence
tableauPersonnages[0] = null;
```

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

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

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

// supprime la première référence
personnage = null;

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

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

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

A retenir

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

Bonnes pratiques

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

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

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

Nous préférerons donc le code suivant :

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

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

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

{

}
```

A cette écriture non optimisée :

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

for ( var i:int = 0; i< tableauDonnees.length; i++ )

{

}
```

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

Nous préférerons l'écriture suivante :

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

var lng:int = tableauDonnees.length;

// déclaration de la variable elementEnCours une seule et unique fois
var elementEnCours:int;

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

{

    elementEnCours = tableauDonnees[i];

}
```

A l'écriture suivante non optimisée :

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

for ( var i:int = 0; i< tableauDonnees.length; i++ )

{

    // déclaration de la variable elementEnCours à chaque itération
    var elementEnCours:int = tableauDonnees[i];

}
```

Découvrons à présent quelques dernières subtilités du langage ActionScript 3.

Autres subtilités

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

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

```
function initilisation ( ):Void

{


}
```

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

```
function initilisation ( ):void

{


}
```

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

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

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

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

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

for each ( var p:* in personnage )

{

    /* affiche :
    50
    Bobby
    */
    trace( p );

}
```

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

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

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

for each ( var p:Boolean in personnage )

{

    /* affiche :
    true
    true
    */
    trace( p );
```

```
}
```

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

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

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

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

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

Vous êtes prêt pour la grande aventure ActionScript 3 ?