Formation à ACTIONSCRIPT® 3.0

Informations juridiques

Vous trouverez des informations juridiques à l'adresse http://help.adobe.com/fr_FR/legalnotices/index.html.

Sommaire

Chapitre 1 : Introduction à ActionScript 3.0

A propos d'ActionScript

ActionScript est le langage de programmation des environnements d'exécution d'Adobe® Flash® Player et Adobe® AIR™. Il assure l'interactivité, le traitement des données et bien d'autres fonctions pour le contenu Flash, Flex et AIR et les applications associées.

ActionScript s'exécute dans la machine virtuelle ActionScript (AVM), un composant de Flash Player et AIR. Le code ActionScript est généralement compilé en pseudo-code (sorte de langage de programmation écrit et compris par les ordinateurs) par un compilateur, tel celui intégré à Adobe® Flash® Professional ou celui intégré à Adobe® Flash® Builder™ et fourni dans le kit de développement SDK d'Adobe® Flex™. Le pseudo-code est intégré aux fichiers SWF, qui sont exécutés par Flash Player et AIR.

ActionScript 3.0 constitue un modèle de programmation solide, bien connu des développeurs possédant des connaissances élémentaires sur la programmation orientée objets. Parmi les principales fonctionnalités d'ActionScript 3.0 qui ont été améliorées par rapport aux versions antérieures d'ActionScript figurent :

- Une nouvelle machine virtuelle ActionScript, appelée AVM2, qui exploite un nouveau jeu d'instructions de pseudo-code binaire et améliore grandement les performances.
- Un code de compilateur plus moderne qui effectue des optimisations de plus bas niveau que les versions antérieures du compilateur.
- Une interface de programmation (API) étendue et améliorée, avec contrôle de bas niveau des objets et un véritable modèle orienté objet.
- Une API XML reposant sur la spécification ECMAScript pour XML (E4X) (ECMA-357 niveau 2). E4X est une extension de langage d'ECMAScript qui ajoute XML comme type de données natif.
- Un modèle d'événements fondé sur la spécification d'événements du modèle d'objet de document (DOM, Document Object Model) niveau 3.

Avantages d'ActionScript 3.0

Les possibilités d'ActionScript 3.0 dépassent largement les fonctions de programmation des versions précédentes. Cette version est conçue pour faciliter la création d'applications très complexes impliquant d'importants jeux de données et des bases de code orientées objet et réutilisables. Si ActionScript 3.0 n'est pas indispensable à l'exécution de contenu dans Adobe Flash Player, il ouvre néanmoins la voie à des améliorations de performance uniquement disponibles dans AVM2 (la machine virtuelle d'ActionScript 3.0). Le code d'ActionScript 3.0 peut s'exécuter jusqu'à dix fois plus vite que le code des versions antérieures d'ActionScript.

L'ancienne version de la machine virtuelle, AVM1, exécute le code ActionScript 1.0 et ActionScript 2.0. Elle est prise en charge par Flash Player 9 et 10 pour assurer la compatibilité ascendante avec le contenu existant.

Nouveautés d'ActionScript 3.0

Bien que de nombreuses classes et fonctions d'ActionScript 3.0 s'apparentent à celles d'ActionScript 1.0 et 2.0, son architecture et sa conceptualisation diffèrent des versions précédentes. Parmi les améliorations d'ActionScript 3.0, on compte de nouvelles fonctions du langage de base et une API avancée, qui accroît le contrôle des objets de bas niveau.

Fonctions du langage de base

Le langage de base définit les éléments de construction fondamentaux du langage de programmation, par exemple les arguments, expressions, conditions, boucles et types. ActionScript 3.0 contient de nombreuses fonctions qui accélèrent le processus de développement.

Exceptions d'exécution

ActionScript 3.0 peut signaler davantage de conditions d'erreur que les versions précédentes. Utilisées pour les conditions d'erreur courantes, les exceptions d'exécution améliorent la procédure de débogage et vous permettent de développer des applications susceptibles de gérer les erreurs de manière fiable. Les erreurs d'exécution peuvent fournir des traces de pile qui identifient le fichier source et le numéro de ligne, pour un repérage plus rapide des erreurs.

Types d'exécution

Dans ActionScript 3.0, les informations de type sont préservées lors de l'exécution. Elles permettent de vérifier les types lors de l'exécution, optimisant ainsi l'intégrité des types du système. Les informations de type servent également à représenter les variables dans les représentations machine natives, ce qui accroît les performances et réduit l'utilisation de la mémoire. Dans ActionScript 2.0, les annotations de type visent avant tout à aider le développeur ; lors de l'exécution, toutes les valeurs sont typées dynamiquement.

Classes scellées

ActionScript 3.0 introduit le concept de classe scellée. Une telle classe possède uniquement un jeu fixe de propriétés et de méthodes, définies lors de la compilation. Il est impossible d'en ajouter d'autres. Ainsi, la vérification effectuée au moment de la compilation est plus stricte et garantit une plus grande robustesse des programmes. L'utilisation de la mémoire est également optimisée puisqu'une table de hachage interne n'est pas requise pour chaque occurrence d'objet. Les classes dynamiques sont également disponibles par le biais du mot-clé `dynamic`. Bien que scellées par défaut, toutes les classes d'ActionScript 3.0 peuvent être déclarées dynamiques grâce au mot-clé `dynamic`.

Fermetures de méthodes

ActionScript 3.0 permet l'utilisation d'une fermeture de méthode qui se rappelle automatiquement l'occurrence de son objet d'origine. Cette fonction s'avère utile dans le traitement des événements. Dans ActionScript 2.0, les fermetures de méthode ne gardent pas la trace de l'occurrence d'objet à partir de laquelle elles ont été extraites, d'où un comportement inattendu lors de l'appel de la fermeture de méthode.

ECMAScript pour XML (E4X)

ActionScript 3.0 intègre ECMAScript pour XML (E4X), récemment normalisé sous le nom ECMA-357. E4X offre un jeu d'éléments de langage naturels et courants qui permettent de manipuler XML. Contrairement aux API classiques d'analyse XML, XML et E4X fonctionnent comme un type de données natif du langage. E4X simplifie le développement d'applications exploitant XML grâce à une réduction drastique du volume de code requis.

Pour visualiser la spécification E4X d'ECMA, aller à www.ecma-international.org (disponible en anglais uniquement).

Expressions régulières
ActionScript 3.0 inclut une prise en charge native des expressions régulières afin d'accélérer la recherche et la manipulation des chaînes. Dans ActionScript 3.0, cette prise en charge suit la version 3 de la spécification de langage ECMAScript (ECMA-262).

Espaces de noms
Les espaces de noms sont semblables aux spécificateurs d'accès classiques qui assurent le contrôle de visibilité des déclarations (`public`, `private`, `protected`). Ils fonctionnent comme des spécificateurs d'accès personnalisés, qui portent le nom de votre choix. Les espaces de noms sont dotés d'un identifiant de ressource universel (URI, Universal Resource Identifier) afin d'éviter les collisions. Ils servent également à représenter les espaces de noms XML en cas d'utilisation d'E4X.

Nouveaux types de primitives
ActionScript 3.0 comprend trois types numériques : Number, int et uint. Number représente un nombre en virgule flottante à deux décimales. Le type int est un entier signé 32 bits qui permet au code ActionScript de profiter de la rapidité de traitement mathématique de l'unité centrale. Il s'avère pratique pour les compteurs de boucles et les variables utilisant des entiers. Le type uint est un type d'entier non signé 32 bits, utile pour les valeurs de couleurs RVB, les compteurs d'octets, etc. ActionScript 2.0, en revanche, utilise un seul type numérique, Number.

Fonctions API

Les API d'ActionScript 3.0 contiennent un grand nombre de classes qui vous permettent de contrôler les objets de bas niveau. L'architecture du langage est conçue pour être plus intuitive que les versions antérieures. Ces nouvelles classes étant trop nombreuses pour autoriser une présentation détaillée à ce stade, il est utile de mettre en avant quelques changements significatifs.

Modèle d'événements DOM3
Le modèle d'événements Document Object Model de niveau 3 (DOM3) offre une méthode standard de génération et de traitement des messages d'événement. Il permet aux objets composant les applications d'interagir et de communiquer tout en conservant leur état et en réagissant aux changements. Etabli à partir des spécifications d'événements DOM niveau 3 du World Wide Web Consortium, ce modèle fournit un mécanisme plus clair et plus efficace que les systèmes d'événements disponibles dans les versions antérieures d'ActionScript.

Les événements et événements d'erreur se trouvent dans le package flash.events. Les composants Flash Professional et la structure Flex utilisant le même modèle d'événements, le système d'événements est unifié sur l'ensemble de la plate-forme Flash.

API de liste d'affichage
L'API d'accès à la liste d'affichage, c'est-à-dire l'arborescence contenant tous les éléments visuels de l'application, est constituée de classes permettant de manipuler les primitives visuelles.

La classe Sprite est un élément de construction léger, adaptée à la classe de base des éléments visuels tels que les composants d'interface. La classe Shape représente des formes vectorielles brutes. Il est possible d'instancier ces classes naturellement à l'aide de l'opérateur `new`, mais aussi de les redéfinir dynamiquement comme parent à tout moment.

La gestion de profondeur est automatique. Des méthodes permettent de spécifier et de gérer l'ordre de superposition des objets.

Gestion des données et contenus dynamiques

ActionScript 3.0 comprend des mécanismes de chargement et de gestion des actifs et des données au sein de l'application qui se caractérisent par leur intuitivité et leur cohérence dans l'ensemble de l'API. La classe Loader propose un unique mécanisme de chargement des fichiers SWF et des actifs d'image, et permet d'accéder à des informations détaillées sur le contenu chargé. La classe URLLoader offre un mécanisme distinct de chargement du texte et des données binaires dans les applications orientées données. La classe Socket permet la lecture et l'écriture des données binaires dans les sockets de serveur, quel que soit le format.

Accès aux données de bas niveau

Diverses API permettent d'accéder à des données de bas niveau. Pour le téléchargement de données, la classe URLStream donne accès aux données sous forme binaire brute pendant le téléchargement. Avec la classe ByteArray, vous pouvez optimiser la lecture, l'écriture et la manipulation des données binaires. L'API Sound assure le contrôle précis du son par le biais des classes SoundChannel et SoundMixer. Des API liées à la sécurité fournissent des informations sur les droits de sécurité d'un fichier SWF ou du contenu chargé, pour une gestion plus efficace des erreurs de sécurité.

Utilisation de texte

ActionScript 3.0 contient un package flash.text destiné à l'ensemble des API relatives au texte. La classe TextLineMetrics propose des mesures détaillées relatives à une ligne de texte au sein d'un champ de texte. Elle remplace la méthode `TextFormat.getTextExtent()` d'ActionScript 2.0. La classe TextField contient diverses méthodes de bas niveau, qui fournissent des informations déterminées sur une ligne de texte ou un caractère unique dans un champ de texte. Par exemple, la méthode `getCharBoundaries()` renvoie un rectangle représentant le cadre de sélection d'un caractère. La méthode `getCharIndexAtPoint()` renvoie l'index d'un caractère à un point donné. La méthode `getFirstCharInParagraph()` renvoie l'index du premier caractère d'un paragraphe. Les méthodes de niveau de ligne incluent `getLineLength()`, qui renvoie le nombre de caractères d'une ligne de texte donnée, et `getLineText()`, qui renvoie le texte de la ligne spécifiée. La classe Font permet de gérer les polices intégrées à des fichiers SWF.

Les classes du package flash.text.engine, qui constituent Flash Text Engine, sont conçues pour un contrôle de bas niveau du texte et permettent de créer des structures et des composants texte.

Chapitre 2 : Prise en main d'ActionScript

Concepts de programmation de base

ActionScript étant un langage de programmation, il vous sera plus facile de l'apprendre si vous maîtrisez déjà quelques concepts généraux de programmation.

Quel est le rôle d'un programme informatique ?

Pour commencer, il est intéressant d'avoir une idée conceptuelle de la nature et du rôle d'un programme informatique. Celui-ci présente deux aspects principaux :

- Il est constitué d'une série d'instructions ou d'étapes que l'ordinateur doit effectuer.
- Chaque étape implique à terme la manipulation d'informations ou de données.

En fait, un programme informatique n'est rien d'autre qu'une liste d'actions que vous demandez à l'ordinateur d'exécuter l'une après l'autre. Chacune de ces demandes d'exécution d'action s'appelle une *instruction*. Dans ActionScript, chaque instruction se termine par un point-virgule.

Par nature, le seul rôle d'une instruction de programme consiste à manipuler quelques données stockées dans la mémoire de l'ordinateur. Prenons un exemple simple : vous indiquez à l'ordinateur de faire la somme de deux nombres et de stocker le résultat en mémoire. Dans un cas de figure plus compliqué, imaginez un rectangle dessiné sur l'écran ; vous rédigez un programme pour le déplacer à un autre emplacement de l'écran. L'ordinateur mémorise certaines informations relatives au rectangle : les coordonnées x, y de sa position, sa largeur et sa hauteur, sa couleur, etc. Chacune de ces informations est stockée dans la mémoire de l'ordinateur. Un programme de déplacement du rectangle comporterait des étapes telles que « régler la coordonnée x sur 200 ; régler la coordonnée y sur 150. » En d'autres termes, il définirait de nouvelles valeurs pour les coordonnées x et y. En arrière-plan, l'ordinateur manipule ces données pour convertir les nombres en images visibles à l'écran. Il suffit cependant de savoir que le processus de « déplacement d'un rectangle à l'écran » implique uniquement la modification de données dans la mémoire de l'ordinateur.

Variables et constantes

La programmation consiste principalement à modifier des informations dans la mémoire de l'ordinateur. Il est donc important de disposer d'un moyen permettant de représenter une information unique dans un programme. Une *variable* est un nom qui représente une valeur dans la mémoire de l'ordinateur. Lorsque vous rédigez des instructions visant à manipuler des valeurs, vous écrivez le nom de la variable plutôt que la valeur. Chaque fois que l'ordinateur rencontre le nom de la variable dans le programme, il cherche dans sa mémoire la valeur à utiliser. Si vous disposez par exemple de deux variables appelées `value1` et `value2`, chacune contenant un nombre, vous pouvez additionner ces deux nombres à l'aide de l'instruction suivante :

```
value1 + value2
```

Lorsqu'il exécute véritablement la procédure, l'ordinateur recherche les valeurs correspondant à chaque variable et les ajoute.

Dans ActionScript 3.0, une variable se compose de trois éléments :

- Le nom de la variable
- Le type de données qui peut être stocké dans la variable

- La valeur réelle stockée dans la mémoire de l'ordinateur

Nous venons de voir comment l'ordinateur utilise le nom comme un espace réservé destiné à la valeur. Le type de données a également une importance. Lorsque vous créez une variable dans ActionScript, vous indiquez le type spécifique de données auquel elle est réservée. Les instructions du programme peuvent alors uniquement stocker ce type de données dans la variable. Vous pouvez manipuler la valeur à l'aide des caractéristiques particulières associées à ce type de données. Dans ActionScript, la création d'une variable (on parle également de *déclaration* de variable) s'effectue à l'aide de l'instruction `var` :

```
var value1:Number;
```

Dans cet exemple, nous indiquons à l'ordinateur de créer la variable `value1`, qui peut uniquement contenir des données Number, un type de données spécialement défini dans ActionScript. Il est également possible de stocker immédiatement une valeur dans la variable :

```
var value2:Number = 17;
```

Adobe Flash Professional

Flash Professional propose une autre méthode de déclaration des variables. Lorsque vous placez un symbole de clip, un symbole de bouton ou un champ de texte sur la scène, vous pouvez lui attribuer un nom d'occurrence dans l'Inspecteur des propriétés. En arrière-plan, Flash Professional crée une variable du même nom que celui de l'occurrence. Vous pouvez utiliser ce nom dans votre code ActionScript pour représenter cet élément de la scène. Par exemple, si un symbole de clip se trouve sur la scène et que vous lui attribuez le nom `rocketShip`, chaque fois que vous utilisez la variable `rocketShip` dans le code ActionScript, c'est en fait ce clip que vous manipulez.

Une *constante* s'apparente à une variable, dans la mesure où elle correspond à un nom représentant une valeur dans la mémoire de l'ordinateur et est associée à un type de données spécifique. En revanche, vous ne pouvez affecter qu'une seule valeur à une constante dans une application ActionScript. Une fois affectée à une constante, une valeur reste la même dans l'ensemble de l'application. La syntaxe de déclaration d'une constante est identique à celle de la déclaration d'une variable, à la différence près que vous substituez le mot-clé `const` au mot-clé `var` :

```
const SALES_TAX_RATE:Number = 0.07;
```

Une constante permet de définir une valeur qui s'utilise à plusieurs emplacements dans un projet et reste identique dans des circonstances normales. L'utilisation d'une constante plutôt que d'une valeur littérale améliore la lisibilité de votre code. Considérons par exemple deux versions d'un même code. L'une multiplie un prix par `SALES_TAX_RATE`, tandis que l'autre le multiplie par `0.07`. La version contenant la constante `SALES_TAX_RATE` est plus facile à comprendre. Imaginons, en outre, que la valeur définie par la constante change. Si vous représentez cette valeur par une constante dans l'ensemble du projet, il vous suffit d'intervenir à un seul emplacement (dans la déclaration de la constante), alors que si vous utilisez des valeurs littérales codées en dur, vous devez changer chaque occurrence.

Types de données

Dans ActionScript, de nombreux types de données sont à votre disposition pour la création de variables. Certains d'entre eux peuvent être considérés comme « simples » ou « fondamentaux » :

- String : une valeur textuelle, telle qu'un nom ou le texte d'un chapitre de livre
- Numeric : ActionScript 3.0 inclut trois types de données spécifiques aux valeurs numériques :
  - Number : toute valeur numérique, y compris les valeurs avec ou sans fraction
  - int : un nombre entier (sans fraction)
  - uint : un nombre entier « non signé », c'est-à-dire, un nombre entier qui ne peut pas être négatif

- Boolean : une valeur vrai/faux, qui indique par exemple si une instruction Switch est active ou si deux valeurs sont égales

Les types de données simples représentent une information unique : un nombre ou une séquence de texte, par exemple. Cependant, la majorité des types de données définis dans ActionScript sont complexes car ils représentent un ensemble de valeurs regroupées dans un même conteneur. Par exemple, une variable du type Date représente une valeur unique (un point temporel). Toutefois, la valeur date se compose en réalité de plusieurs valeurs (le jour, le mois, l'année, les heures, les minutes, les secondes, etc.), qui correspondent elles-mêmes à des nombres individuels. Ainsi, bien que nous percevions une date comme une valeur unique (et qu'il soit possible de la traiter comme telle en créant une variable Date), l'ordinateur, en interne, la considère comme un groupe de valeurs qui, ensemble, définissent une seule date.

La plupart des types de données intégrés et ceux définis par les programmeurs sont des types de données complexes. Voici quelques exemples de types de données complexes que vous connaissez probablement :

- MovieClip : un symbole de clip
- TextField : un champ de texte dynamique ou saisi
- SimpleButton : un symbole de bouton
- Date : une information relative à un point temporel (date et heure)

Deux termes sont souvent utilisés comme synonymes de type de données : classe et objet. Une *classe* est tout simplement la définition d'un type de données ; un modèle, en quelque sorte, s'appliquant à tous les objets du type de données et qui revient à dire « toutes les variables du type de données Exemple sont dotées des caractéristiques suivantes : A, B et C ». Un *objet*, quant à lui, est une occurrence réelle d'une classe. Une variable dont le type de données correspond à MovieClip, par exemple, peut être décrite comme un objet MovieClip. Les exemples ci-après décrivent la même chose :

- Le type de données de la variable `myVariable` est Number.
- La variable `myVariable` est une occurrence de Number.
- La variable `myVariable` est un objet Number.
- La variable `myVariable` est une occurrence de la classe Number.

Utilisation des objets

ActionScript constitue ce que l'on appelle un langage de programmation orienté objet. Un langage de programmation orienté objet est une simple approche de la programmation, rien d'autre qu'une manière d'organiser le code d'un programme à l'aide d'objets.

Nous avons défini plus haut un « programme informatique » comme une série de procédures ou d'instructions que l'ordinateur effectue. Nous pouvons alors considérer qu'un programme informatique n'est qu'une longue liste d'instructions. Dans le cas de la programmation orientée objets, cependant, les instructions du programme se divisent en différents objets : le code étant rassemblé en groupes de fonctionnalités, un même conteneur réunit des types de fonctionnalités et des informations connexes.

Adobe Flash Professional

Si vous avez travaillé avec des symboles dans Flash Professional, vous savez déjà manipuler les objets. Imaginons que vous ayez défini un symbole de clip (le dessin d'un rectangle, par exemple) et vous en ayez placé une copie sur la scène. Ce symbole de clip constitue aussi un objet (au sens littéral) dans ActionScript ; il s'agit d'une occurrence de la classe MovieClip.

Vous avez la possibilité de modifier plusieurs caractéristiques du clip. Lorsqu'il est sélectionné, vous pouvez modifier certaines valeurs de l'Inspecteur des propriétés, telles que la coordonnée x ou la largeur. Vous pouvez également effectuer différents réglages de couleur comme la transparence alpha, ou encore lui appliquer un filtre d'ombre portée. D'autres outils Flash Professional permettent d'effectuer davantage de modifications, par exemple l'outil Transformation libre pour faire pivoter le rectangle. Toutes ces actions de transformation du symbole de clip disponibles dans Flash Professional le sont également dans ActionScript. Pour les utiliser, vous devez modifier les données réunies dans un ensemble appelé objet MovieClip.

Dans la programmation orientée objets que propose ActionScript, chaque classe comprend trois types de caractéristiques :

- Propriétés
- Méthodes
- Evénements

Ces éléments servent à gérer les données que le programme utilise, et à déterminer les actions à exécuter ainsi que l'ordre d'exécution.

Propriétés

Une propriété représente l'une des données réunies dans un objet. Un objet song, par exemple, peut présenter des propriétés appelées `artist` et `title`. La classe MovieClip possède des propriétés telles que `rotation`, `x`, `width` et `alpha`. Les propriétés s'utilisent comme des variables individuelles. On pourrait même les envisager comme les variables « enfant » d'un objet.

Voici des exemples de code ActionScript utilisant des propriétés. Cette ligne de code déplace l'objet MovieClip nommé `square` vers la coordonnée x de 100 pixels :

```
square.x = 100;
```

Le code ci-après utilise la propriété rotation pour faire pivoter le MovieClip `square` de manière à lui donner la même orientation que le MovieClip `triangle` :

```
square.rotation = triangle.rotation;
```

Ce code modifie l'échelle horizontale du MovieClip `square` pour qu'il soit une fois et demie plus large que précédemment :

```
square.scaleX = 1.5;
```

Observez la structure commune : vous utilisez une variable (`square`, `triangle`) comme nom de l'objet, suivi d'un point() puis du nom de la propriété (`x`, `rotation`, `scaleX`). Le point, ou *opérateur point*, sert à indiquer que vous accédez à l'un des éléments enfant d'un objet. La structure dans son ensemble, « nom de variable-point-nom de propriété » est utilisée telle une variable, comme le nom d'une valeur unique dans la mémoire de l'ordinateur.

Méthodes

Une *méthode* est une action qui peut être effectuée par un objet. Par exemple, si vous avez élaboré, dans Flash Professional, un symbole de clip dont le scénario contient plusieurs images clés et animations, ce clip peut être lu ou arrêté, ou recevoir l'instruction de placer la tête de lecture sur une image donnée.

Le code ci-dessous indique au MovieClip nommé `shortFilm` de commencer la lecture :

```
shortFilm.play();
```

Cette ligne de code arrête la lecture du MovieClip `shortFilm` (la tête de lecture s'arrête à l'endroit où elle se trouve, comme lorsque vous mettez une vidéo en pause) :

```
shortFilm.stop();
```

Ce code-ci indique au MovieClip `shortFilm` de placer la tête de lecture sur Frame 1 et d'arrêter la lecture (comme si vous rembobiniez une vidéo) :

```
shortFilm.gotoAndStop(1);
```

L'accès aux méthodes, comme pour les propriétés, s'effectue en écrivant le nom de l'objet (une variable) suivi d'un point puis du nom de la méthode et de parenthèses. Les parenthèses servent à indiquer que vous *appelez* la méthode, c'est-à-dire, que vous demandez à l'objet d'effectuer une action. Il arrive que des valeurs (ou variables) soient placées dans ces parenthèses de manière à transmettre des informations supplémentaires nécessaires à l'exécution de l'action. On appelle ces valeurs des *paramètres* de méthode. Par exemple, la méthode `gotoAndStop()` doit savoir quelle image atteindre ; un paramètre est donc requis dans les parenthèses. D'autres méthodes, telles `play()` et `stop()`, ont une signification univoque et n'ont donc besoin d'aucune information complémentaire. Elles sont néanmoins suivies de parenthèses.

Contrairement aux propriétés (et aux variables), les méthodes ne servent pas d'espaces réservés. Certaines méthodes peuvent cependant effectuer des calculs et renvoyer des résultats pouvant servir de variables. C'est le cas de la méthode `toString()` de la classe Number, qui convertit une valeur numérique en une représentation textuelle :

```
var numericData:Number = 9;
var textData:String = numericData.toString();
```

Par exemple, vous pouvez utiliser la méthode `toString()` si vous souhaitez afficher la valeur d'une variable Number dans un champ de texte à l'écran. La propriété `text` de la classe TextField (qui représente le contenu textuel réel affiché à l'écran) se définit comme une chaîne (String), de sorte qu'elle ne puisse contenir que des valeurs textuelles. Cette ligne de code convertit la valeur numérique de la variable `numericData` en texte, puis la fait apparaître à l'écran dans l'objet TextField nommé `calculatorDisplay` :

```
calculatorDisplay.text = numericData.toString();
```

Evénements

Un programme informatique est une série d'instructions que l'ordinateur exécute l'une après l'autre. Certains programmes très simples ne sont rien de plus : l'ordinateur exécute quelques procédures, puis le programme se termine. Toutefois, les programmes ActionScript sont conçus pour poursuivre leur exécution, dans l'attente d'une saisie utilisateur, par exemple. Les événements sont des mécanismes qui déterminent quelles instructions l'ordinateur doit exécuter et à quel moment.

Par essence, les *événements* sont des faits qui surviennent et auxquels ActionScript peut répondre parce qu'il en est conscient au moment où ils se produisent. De nombreux événements sont liés à l'interaction de l'utilisateur, par exemple un clic sur un bouton ou une pression sur une touche du clavier. Il existe néanmoins d'autres types d'événements. Par exemple, si vous utilisez ActionScript pour charger une image externe, il existe un événement capable de vous prévenir lorsque le chargement de l'image est terminé. Pendant son exécution, un programme ActionScript attend que des événements se produisent et, lorsque c'est le cas, il exécute le code ActionScript que vous avez spécifié en réponse.

Gestion des événements de base

La *gestion des événements* est la technique qui permet de spécifier les actions à exécuter en réponse à des événements particuliers. Lors de l'écriture de code ActionScript en vue de la gestion des événements, trois éléments importants sont à identifier :

- Source de l'événement : quel objet sera concerné par l'événement ? Par exemple, sur quel bouton a eu lieu le clic ou quel est l'objet Loader qui charge l'image ? La source de l'événement est également appelée *cible de l'événement* car elle représente l'objet où l'événement est ciblé (là où l'événement a lieu).
- Evénement : que doit-il se passer, à quel événement voulez-vous répondre? Il est important d'identifier correctement l'événement, car de nombreux objets déclenchent plusieurs événements.
- Réponse : quelles actions doivent être exécutées lorsque l'événement se produit ?

Tout code ActionScript de gestion des événements doit contenir ces trois éléments et respecter la structure de base suivante (les éléments en gras sont des espaces réservés à remplir selon le cas envisagé) :

```
function eventResponse(eventObject:EventType):void
{
    // Actions performed in response to the event go here.
}

eventSource.addEventListener(EventType.EVENT_NAME, eventResponse);
```

Ce code a un double rôle. Tout d'abord, il définit une fonction, qui est une manière de spécifier les actions à exécuter en réponse à l'événement. Ensuite, il appelle la méthode `addEventListener()` de l'objet source, « inscrivant » ainsi la fonction auprès de l'événement spécifié de manière que, dès que l'événement survient, les actions de la fonction aient lieu. Nous allons étudier chacun de ces rôles en détail.

Une *fonction* sert à regrouper des actions sous un nom unique, un raccourci qui vous permet de les exécuter. La fonction est identique à la méthode, à cette exception près qu'elle n'est pas nécessairement associée à une classe particulière (on pourrait d'ailleurs définir la méthode ainsi : une fonction associée à une classe donnée). Lorsque vous créez une fonction de gestion des événements, vous devez choisir le nom de la fonction (dans ce cas `eventResponse`) mais aussi spécifier un paramètre (`eventObject`) dans cet exemple). La spécification d'un paramètre de fonction ressemble à la déclaration de variable ; vous devez dans ce cas aussi indiquer le type de données du paramètre (`EventType`, dans cet exemple).

Chaque type d'événement que vous souhaitez écouter est associé à une classe ActionScript. Le type de données que vous spécifiez pour le paramètre de fonction correspond systématiquement à la classe associée à l'événement auquel vous souhaitez réagir. Par exemple, un événement `click` (déclenché par un clic de souris sur un élément) est associé à la classe MouseEvent. Pour écrire une fonction d'écouteur pour un événement `click`, vous lui associez un paramètre de type MouseEvent. Enfin, entre les accolades d'ouverture et de fermeture (`{ ... }`), vous placez les instructions que l'ordinateur doit exécuter lorsque l'événement a lieu.

La fonction de gestion de l'événement est écrite. Vous indiquez ensuite à l'objet source (celui qui provoque l'événement, par exemple le bouton) que cette fonction doit être appelée lorsque l'événement survient. Pour ce faire, vous appelez la méthode `addEventListener()` de cet objet (tous les objets liés à des événements ont également une méthode`addEventListener()`). La méthode `addEventListener()` réclame deux paramètres :

- Tout d'abord, le nom de l'événement auquel vous voulez répondre. Chaque événement est affilié à une classe spécifique pour laquelle est définie une valeur propre à chacun d'eux (en quelque sorte le nom unique de l'événement). Cette valeur sert de premier paramètre.
- Vient ensuite le nom de votre fonction de réponse à l'événement. Sachez qu'un nom de fonction est écrit sans parenthèses lors du transfert en tant que paramètre.

Processus de gestion d'événement

Vous trouverez ci-dessous une description détaillée du processus ayant lieu lorsque vous créez un écouteur d'événements. Dans ce cas, il s'agit d'un exemple illustrant la création d'une fonction d'écouteur appelée lorsque vous cliquez sur un objet `myButton`.

Le code écrit par le programmeur est le suivant :

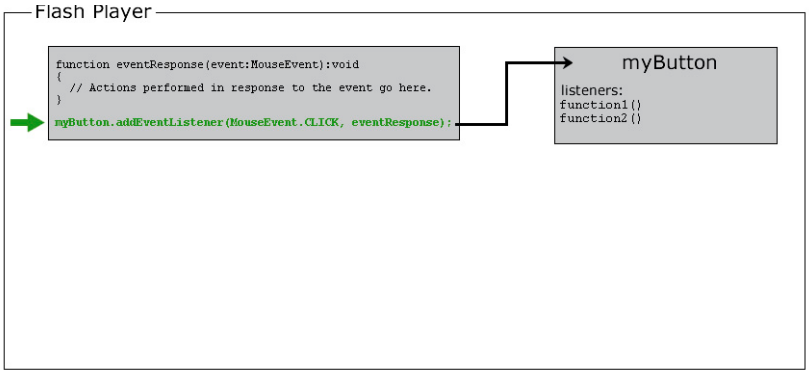
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

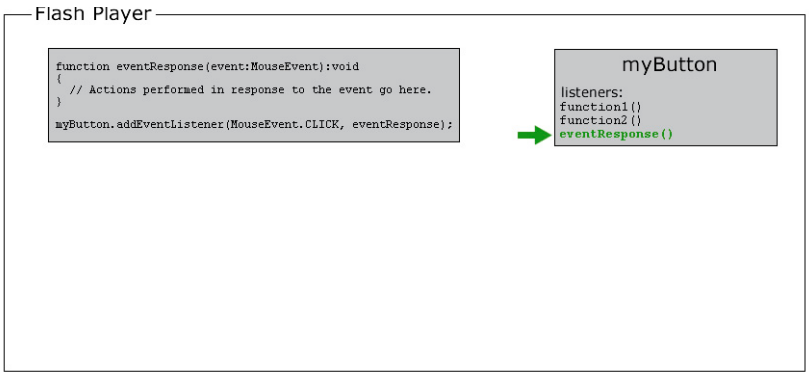
Voici comment ce code devrait fonctionner lorsqu'il est exécuté :

1 Lors du chargement du fichier SWF, l'ordinateur remarque qu'il existe une fonction `eventResponse()`.

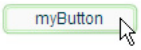
2 Il exécute ensuite le code (notamment les lignes de code qui ne sont pas dans une fonction). Dans ce cas, il s'agit d'une seule ligne de code : l'appel de la méthode `addEventListener()` sur l'objet source de l'événement (`myButton`) et la transmission de la fonction `eventResponse` en tant que paramètre.



En interne, `myButton` a une liste des fonctions qui écoutent chaque événement. Lorsque sa méthode `addEventListener()` est appelée, `myButton` stocke la fonction `eventResponse()` dans sa liste d'écouteurs d'événements.

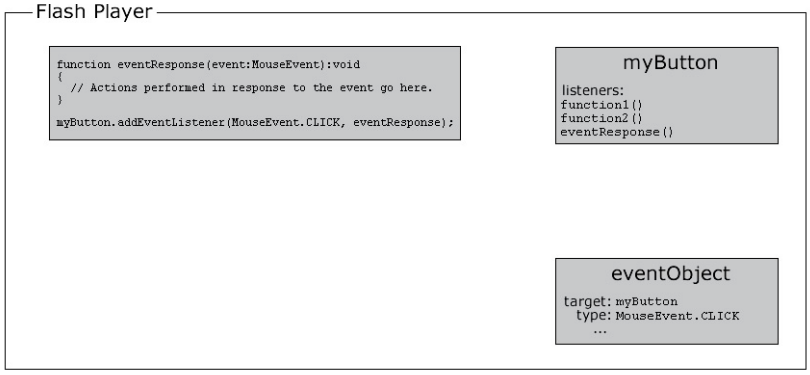


3 A un certain moment, l'utilisateur clique sur l'objet `myButton` et déclenche ainsi son événement `click` (identifié comme `MouseEvent.CLICK` dans le code).

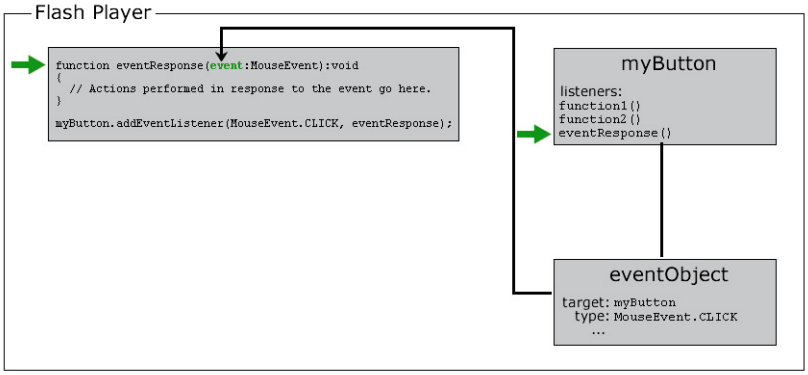


A ce stade, les opérations suivantes ont lieu :

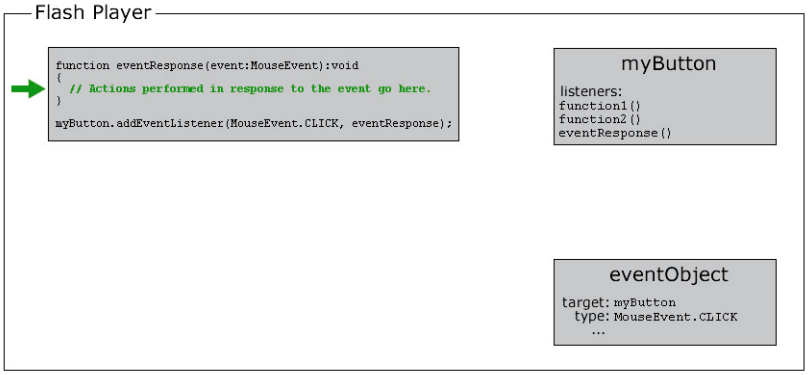
a Un objet est créé, une occurrence de la classe associée à l'événement en question (MouseEvent dans cet exemple). Pour de nombreux événements, il s'agit d'une occurrence de la classe Event ; pour des événements de souris, une occurrence de MouseEvent et pour d'autres événements, une occurrence de la classe qui leur est associée. L'objet créé est appelé l'*objet événement*. Il contient des informations spécifiques sur l'événement qui s'est produit : son type, l'emplacement où il a eu lieu et toute autre information pertinente.



b L'ordinateur consulte ensuite la liste des écouteurs d'événements stockés par `myButton`. Il parcourt ces fonctions l'une après l'autre en les appelant et en transmettant l'objet événement à la fonction en tant que paramètre. Etant donné que la fonction `eventResponse()` est l'un des écouteurs de `myButton`, l'ordinateur appelle la fonction `eventResponse()` dans le cadre de ce processus.



c Lorsque la fonction `eventResponse()` est appelée, le code qu'elle contient est exécuté et vos actions spécifiées sont effectuées.



Exemples de gestion d'événements

Voici quelques exemples plus concrets d'événements qui vous donneront une idée des éléments les plus courants et des variations que vous pourrez utiliser lors de l'écriture de votre propre code de gestion des événements :

- Clic sur un bouton pour lancer la lecture du clip actif. Dans l'exemple suivant, `playButton` est le nom d'occurrence du bouton et `this` est un nom spécial qui signifie « l'objet actif» :

```
this.stop();

function playMovie(event:MouseEvent):void
{
    this.play();
}

playButton.addEventListener(MouseEvent.CLICK, playMovie);
```

- Détection de la saisie dans un champ de texte. Dans cet exemple, `entryText` est un champ de saisie de texte et `outputText` est un champ de texte dynamique :

```
function updateOutput(event:TextEvent):void
{
    var pressedKey:String = event.text;
    outputText.text = "You typed: " + pressedKey;
}

entryText.addEventListener(TextEvent.TEXT_INPUT, updateOutput);
```

- Clic sur un bouton pour atteindre une URL. Dans ce cas, `linkButton` est le nom d'occurrence du bouton :

```
function gotoAdobeSite(event:MouseEvent):void
{
    var adobeURL:URLRequest = new URLRequest("http://www.adobe.com/");
    navigateToURL(adobeURL);
}

linkButton.addEventListener(MouseEvent.CLICK, gotoAdobeSite);
```

Création d'occurrences d'objets

Pour utiliser un objet dans ActionScript, cet objet doit exister. La création d'un objet repose en partie sur la déclaration d'une variable. Toutefois, celle-ci crée uniquement un emplacement vide dans la mémoire de l'ordinateur. Avant d'utiliser ou de manipuler une variable, attribuez-lui toujours une valeur réelle (en créant un objet et en le stockant dans la variable). Le processus de création d'un objet est appelé *instanciation* de l'objet, soit la création d'une occurrence d'une classe particulière.

La création d'une occurrence d'objet peut passer par une méthode simple qui n'implique pas ActionScript. Dans Flash Professional, placez un symbole de clip, un symbole de bouton ou un champ de texte sur la scène, puis attribuez-lui un nom d'occurrence. L'application déclare automatiquement une variable dotée de ce nom d'occurrence, crée une occurrence d'objet et stocke cet objet dans la variable. Il en va de même dans Flex : vous créez un composant MXML, soit par codage d'une balise MXML, soit en plaçant le composant dans l'éditeur de Flash Builder en mode Création. Lorsque vous lui attribuez un identifiant, celui-ci devient le nom d'une variable ActionScript contenant une occurrence de ce composant.

Toutefois, il n'est pas toujours possible de créer un objet visuellement (notamment dans le cas des objets non visuels). Plusieurs méthodes permettent aussi de créer des occurrences d'objet à l'aide d'ActionScript uniquement.

A partir de plusieurs types de données ActionScript, vous pouvez créer une occurrence en utilisant une *expression littérale*, une valeur écrite directement dans le code ActionScript. Voici quelques exemples :

- Valeur numérique littérale (entrez le nombre directement) :

```
var someNumber:Number = 17.239;
var someNegativeInteger:int = -53;
var someUint:uint = 22;
```

- Valeur de chaîne littérale (entourez le texte de doubles guillemets) :

```
var firstName:String = "George";
var soliloquy:String = "To be or not to be, that is the question...";
```

- Valeur booléenne littérale (utilisez la valeur littérale `true` ou `false`) :

```
var niceWeather:Boolean = true;
var playingOutside:Boolean = false;
```

- Valeur de tableau littérale (placez entre crochets une liste de valeurs séparées par des virgules) :

```
var seasons:Array = ["spring", "summer", "autumn", "winter"];
```

- Valeur XML littérale (entrez le XML directement) :

```
var employee:XML = <employee>
        <firstName>Harold</firstName>
        <lastName>Webster</lastName>
    </employee>;
```

ActionScript définit également des expressions littérales pour les types de données Array, RegExp, Object et Function.

Pour tout type de données, le plus simple consiste à créer une occurrence d'objet à l'aide de l'opérateur `new` et du nom de classe, comme suit :

```
var raceCar:MovieClip = new MovieClip();
var birthday:Date = new Date(2006, 7, 9);
```

Pour cette méthode de création d'un objet à l'aide de l'opérateur `new`, on parle souvent « d'appeler le constructeur de la classe ». Un *constructeur* est une méthode spéciale qui est appelée dans le cadre de la création d'une occurrence de classe. Notez que lorsque vous créez une occurrence ainsi, vous placez des parenthèses après le nom de classe et spécifiez parfois des valeurs de paramètres dans les parenthèses, comme vous le faites lorsque vous appelez une méthode.

Vous pouvez utiliser l'opérateur `new` pour créer une occurrence d'objet, même pour les types de données qui permettent de créer des occurrences avec une expression littérale. Par exemple, ces deux lignes de code produisent le même résultat :

```
var someNumber:Number = 6.33;
var someNumber:Number = new Number(6.33);
```

Il est important de se familiariser avec la méthode `new` *NomClasse*`()` de création d'objets. De nombreux types de données ActionScript ne disposent pas de représentation visuelle. Il est donc impossible de les créer en plaçant un élément sur la scène de Flash Professional ou en mode Création dans l'éditeur MXML de Flash Builder. Dans ce cas, vous pouvez uniquement créer une occurrence dans ActionScript à l'aide de l'opérateur `new`.

Adobe Flash Professional
Dans Flash Professional, l'opérateur `new` peut en outre servir à créer une occurrence d'un symbole de clip défini dans la bibliothèque sans être placé sur la scène.

Voir aussi

Utilisation de tableaux

Utilisation d'expressions régulières

Création d'objets MovieClip à l'aide d'ActionScript

Eléments de programme courants

D'autres éléments de construction peuvent servir à la création d'un programme ActionScript.

Opérateurs

Les *opérateurs* sont des symboles spéciaux (et parfois des mots) qui permettent d'effectuer des calculs. Ils sont surtout utilisés dans les opérations mathématiques et la comparaison de valeurs. En règle générale, un opérateur utilise une ou plusieurs valeurs et « établit » un résultat unique. Exemple :

- L'opérateur de somme (`+`) ajoute deux valeurs pour obtenir un nombre unique :

  ```
  var sum:Number = 23 + 32;
  ```

- L'opérateur de multiplication (`*`) multiplie une valeur par une autre pour obtenir un nombre unique :

  ```
  var energy:Number = mass * speedOfLight * speedOfLight;
  ```

- L'opérateur d'égalité (`==`) compare deux valeurs pour vérifier si elles sont égales, afin d'obtenir une valeur vrai/faux (booléenne) unique :

  ```
  if (dayOfWeek == "Wednesday")
  {
      takeOutTrash();
  }
  ```

  Comme l'indique la figure ci-dessus, l'opérateur d'égalité et les autres opérateurs de « comparaison » sont le plus souvent utilisés avec l'instruction `if`afin de déterminer si certaines actions sont effectuées ou non.

Commentaires

Lors de la rédaction du code ActionScript, il peut s'avérer utile de conserver des notes personnelles, expliquant par exemple le fonctionnement de certaines lignes de code ou la raison pour laquelle vous avez fait tel ou tel choix. Les *commentaires de code* permettent d'insérer dans du code du texte que l'ordinateur devra ignorer. ActionScript comprend deux types de commentaires :

- Commentaire sur une ligne : un commentaire sur une ligne est signalé par l'insertion de deux barres obliques en un emplacement quelconque d'une ligne. Tout ce qui apparaît après les barres obliques et jusqu'à la fin de la ligne est ignoré par l'ordinateur :

  ```
  // This is a comment; it's ignored by the computer.
  var age:Number = 10; // Set the age to 10 by default.
  ```

- Commentaire sur plusieurs lignes : un commentaire sur plusieurs lignes comprend un marqueur de début (`/*`), le commentaire lui-même, puis un marqueur de fin de commentaire (`*/`). L'ordinateur ignore tout ce qui apparaît entre les marqueurs de début et de fin, quel que soit le nombre de lignes utilisé par le commentaire :

  ```
  /*
  This is a long description explaining what a particular
  function is used for or explaining a section of code.

  In any case, the computer ignores these lines.
  */
  ```

Une autre utilité des commentaires est de « désactiver » temporairement une ou plusieurs lignes de code. C'est le cas, par exemple, si vous testez différentes façons d'aboutir à un résultat ou essayez d'identifier pourquoi le code ActionScript ne fonctionne pas comme vous le pensiez.

Contrôle du flux

Dans bien des cas, il vous sera nécessaire de répéter des actions de votre code, d'en effectuer certaines et pas d'autres, de réaliser des actions de remplacement selon les conditions rencontrées, etc. Le *contrôle de flux* permet de maîtriser les actions exécutées. ActionScript propose plusieurs types d'éléments de contrôle de flux.

- Fonctions : les fonctions sont comme des raccourcis, elles permettent de regrouper sous un même nom une série d'actions qui serviront à des calculs. Essentielles à la gestion des événements, elles constituent en outre un outil générique de regroupement des instructions.
- Boucles : les structures en boucle permettent de désigner un jeu d'instructions que l'ordinateur doit exécuter un nombre défini de fois ou jusqu'à ce qu'une condition change. Souvent, les boucles sont utilisées pour manipuler plusieurs éléments connexes à l'aide d'une variable dont la valeur change à chaque fois que l'ordinateur achève une boucle.
- Instructions conditionnelles : les instructions conditionnelles permettent de désigner certaines actions à effectuer uniquement dans certaines circonstances ou de définir des ensembles d'actions destinés à différentes conditions. L'instruction conditionnelle la plus courante est `if`. L'instruction `if` vérifie la valeur ou l'expression placée dans ses parenthèses. Si le résultat est `true`, les lignes de code entre accolades sont exécutées ; dans le cas contraire, elles sont ignorées. Exemple :

  ```
  if (age < 20)
  {
      // show special teenager-targeted content
  }
  ```

  Associée à l'instruction `else`, l'instruction `if` permet de désigner les actions à effectuer si la condition n'est pas vérifiée (`true`) :

  ```
  if (username == "admin")
  {
      // do some administrator-only things, like showing extra options
  }
  else
  {
      // do some non-administrator things
  }
  ```

Exemple : Elément de portfolio d'animation (Flash Professional)

Cet exemple indique comment vous pouvez assembler des éléments d'ActionScript dans une application complète. L'élément de portfolio d'animation est un exemple de la façon dont vous pourriez ajouter à une animation linéaire existante des éléments interactifs mineurs. Vous pourriez, par exemple, incorporer une animation créée pour un client dans un portfolio en ligne. Les éléments interactifs à ajouter sont deux boutons sur lesquels l'utilisateur peut cliquer : un pour lancer l'animation et un pour accéder à une URL distincte (telle que le menu du portfolio ou la page d'accueil de l'auteur).

Le processus de création de cet élément peut être divisé en quatre sections principales :

1. Préparer le fichier FLA pour ajouter des éléments ActionScript interactifs
2. Créer et ajouter les boutons
3. Ecrire le code ActionScript

4 Test de l'application.

Préparation à l'ajout d'interactivité

Avant d'ajouter des éléments interactifs à l'animation, nous devons configurer le fichier FLA en créant des emplacements pour ajouter le nouveau contenu. En l'occurrence, il s'agit de créer un espace sur la scène où les boutons sont placés, et un « espace » dans le fichier FLA pour garder différents éléments séparés.

Pour configurer le fichier FLA et ajouter des éléments interactifs :

1 Créez un fichier FLA comportant une animation simple (une interpolation de mouvement simple ou une interpolation de forme, par exemple). Si vous disposez déjà d'un fichier FLA contenant l'animation que vous présentez dans le projet, ouvrez-le et enregistrez-le sous un nouveau nom.

2 Choisissez l'endroit où vous souhaitez que les deux boutons apparaissent à l'écran. L'un d'eux lance l'animation et l'autre effectue un lien vers le portfolio de l'auteur ou la page d'accueil. Si nécessaire, libérez ou ajoutez de l'espace sur la scène pour ce nouveau contenu. Le cas échéant, vous pouvez créer un écran de démarrage sur la première image, auquel cas, décalez l'animation afin qu'elle démarre sur l'image 2 ou ultérieurement.

3 Ajoutez un nouveau calque, au-dessus des autres dans le scénario, et nommez-le **buttons**. Il s'agit du calque auquel vous ajouterez les boutons.

4 Ajoutez un nouveau calque, au-dessus du calque buttons, et nommez-le **actions**. C'est là que vous ajouterez le code ActionScript à votre application.

Création et ajout de boutons

Vous allez à présent créer et positionner les boutons qui constituent le centre de l'application interactive.

Pour créer et ajouter des boutons au fichier FLA :

1 A l'aide des outils de dessin, créez l'aspect visuel de votre premier bouton (celui de lecture) sur le calque buttons. Par exemple, dessinez un ovale horizontal avec du texte par-dessus.

2 A l'aide de l'outil de sélection, sélectionnez toutes les parties graphiques du bouton.

3 Dans le menu principal, choisissez Modifier > Convertir en symbole.

4 Dans la boîte de dialogue, choisissez le type de symbole de bouton, donnez-lui un nom et cliquez sur OK.

5 Le bouton étant sélectionné, dans l'Inspecteur des propriétés, affectez-lui le nom d'occurrence **playButton**.

6 Répétez les étapes 1 à 5 afin de créer le bouton qui permettra à l'utilisateur d'accéder à la page d'accueil de l'auteur. Nommez ce bouton **homeButton**.

Ecriture du code

Le code ActionScript de cette application peut être divisé en trois ensembles de fonctionnalités, même s'ils sont tous entrés au même endroit. Le code doit effectuer les trois opérations suivantes :

- Arrêter la tête de lecture dès le chargement du fichier SWF (lorsque la tête de lecture atteint l'image 1).
- Ecouter un événement pour lancer la lecture du fichier SWF lorsque l'utilisateur clique sur le bouton de lecture.
- Ecouter un événement pour que le navigateur accède à l'URL appropriée lorsque l'utilisateur clique sur le bouton de la page d'accueil de l'auteur.

Pour créer un code qui arrête la tête de lecture lorsqu'elle atteint l'image 1 :

1 Sélectionnez l'image-clé sur l'image 1 du calque actions.

2 Pour ouvrir le panneau Actions, sélectionnez Fenêtre > Actions dans le menu principal.

3 Dans le panneau Script, entrez le code suivant :

```
stop();
```

Pour écrire un code qui lance l'animation lorsque l'utilisateur clique sur le bouton de lecture :

1 A la fin du code entré aux étapes précédentes, ajoutez deux lignes vides.

2 Entrez le code suivant en bas du script :

```
function startMovie(event:MouseEvent):void
{
    this.play();
}
```

Ce code définit une fonction appelée `startMovie()`. Lorsque la fonction `startMovie()` est appelée, elle lance la lecture du scénario principal.

3 Sur la ligne qui suit le code ajouté à l'étape précédente, entrez cette ligne de code :

```
playButton.addEventListener(MouseEvent.CLICK, startMovie);
```

Cette ligne de code enregistre la fonction `startMovie()` comme écouteur de l'événement `click` de `playButton`. Ainsi, chaque fois que l'utilisateur clique sur le bouton `playButton`, la fonction `startMovie()` est appelée.

Pour rédiger un code qui permet au navigateur d'accéder à une URL lorsque l'utilisateur clique sur le bouton de la page d'accueil :

1 A la fin du code entré aux étapes précédentes, ajoutez deux lignes vides.

2 Entrez ce code au bas du script :

```
function gotoAuthorPage(event:MouseEvent):void
{
    var targetURL:URLRequest = new URLRequest("http://example.com/");
    navigateToURL(targetURL);
}
```

Ce code définit une fonction `gotoAuthorPage()`. Cette fonction crée d'abord une occurrence d'URLRequest représentant l'URL http://example.com/, puis transmet cette URL à la fonction `navigateToURL()` afin que le navigateur de l'utilisateur l'ouvre.

3 Sur la ligne qui suit le code ajouté à l'étape précédente, entrez cette ligne de code :

```
homeButton.addEventListener(MouseEvent.CLICK, gotoAuthorPage);
```

Cette ligne de code enregistre la fonction `gotoAuthorPage()` comme écouteur pour l'événement `click` de `homeButton`. Ainsi, chaque fois que l'utilisateur clique sur le bouton `homeButton`, la fonction `gotoAuthorPage()` est appelée.

Test de l'application

A ce stade, l'application est entièrement opérationnelle. Testons-la pour nous assurer que c'est le cas.

Pour tester l'application :

1 Dans le menu principal, sélectionnez Contrôle > Tester l'animation. Flash Professional crée le fichier SWF et l'ouvre dans une fenêtre Flash Player.

2 Testez les deux boutons pour vérifier qu'ils fonctionnent.

3 Si ce n'est pas le cas, vérifiez les points suivants :

- Les deux boutons ont-ils des noms d'occurrence différents ?
- Les appels à la méthode `addEventListener()` utilisent-ils les mêmes noms que les noms d'occurrence des boutons ?
- Les noms d'événement corrects sont-ils utilisés dans les appels à la méthode `addEventListener()` ?
- Le paramètre correct est-il spécifié pour chacune des fonctions (qui nécessitent toutes les deux un seul paramètre avec le type de données MouseEvent) ?

Tous ces points et la plupart des autres erreurs possibles entraînent l'apparition d'un message d'erreur lorsque vous choisissez la commande Tester l'animation ou que vous cliquez sur le bouton pendant le test. Recherchez les erreurs de compilation dans le panneau prévu à cet effet (celles qui ont lieu lorsque vous choisissez d'abord Tester l'animation). Recherchez les erreurs d'exécution dans le panneau Sortie. Il s'agit des erreurs qui ont lieu pendant la lecture du contenu, lorsque vous cliquez sur un bouton, par exemple.

Création d'applications avec ActionScript

La création d'une application avec ActionScript nécessite d'autres connaissances que la syntaxe et les noms de classes à utiliser. Bien que la documentation de la plate-forme Flash soit essentiellement axée sur ces deux sujets (la syntaxe et l'utilisation des classes ActionScript), d'autres informations pourront vous être utiles :

- Quels sont les programmes qui permettent d'écrire du code ActionScript ?
- Comment ce code s'organise-t-il ?
- Comment s'intègre-t-il à une application ?
- Quelles étapes faut-il respecter dans le développement d'une application ActionScript ?

Options d'organisation du code

Le code ActionScript 3.0 peut servir à générer de nombreuses applications, qu'il s'agisse d'une simple animation graphique ou d'un système complexe de traitement des transactions client/serveur. Selon le type d'application envisagé, choisissez l'une ou plusieurs des méthodes suivantes pour intégrer ActionScript dans votre projet.

Stockage du code dans les images d'un scénario Flash Professional

Dans Flash Professional, vous pouvez ajouter du code ActionScript à toute image placée dans un scénario. Ce code est exécuté pendant la lecture du clip, au moment où la tête de lecture atteint l'image.

L'insertion de code ActionScript dans des images est une manière simple d'ajouter des comportements à des applications créées dans Flash Professional. Vous pouvez placer du code dans n'importe quelle image du scénario principal ou de celui d'un symbole de clip. Cette souplesse a néanmoins un coût. Lorsque vous créez des applications assez volumineuses, vous risquez de ne plus savoir quelles images contiennent quels scripts. A terme, cela peut compliquer la maintenance de l'application.

Pour simplifier l'organisation de leur code ActionScript dans Flash Professional, de nombreux développeurs placent ce code uniquement dans la première image du scénario ou sur un calque spécifique du document Flash. Il est ainsi plus facile de retrouver et de maintenir le code dans les fichiers FLA Flash. Toutefois, la réutilisation du même code dans un autre projet Flash Professional oblige à copier et coller le code dans le nouveau fichier.

Si vous voulez continuer à pouvoir utiliser votre code ActionScript dans de futurs projets Flash Professional, stockez ce code dans des fichiers ActionScript externes (des fichiers texte dotés de l'extension .as).

Incorporation de code dans les fichiers Flex MXML

Dans un environnement de développement Flex tel que Flash Builder, vous pouvez placer le code ActionScript dans une balise `<fx:Script>` dans un fichier MXML Flex. Dans les projets de grande taille, cette technique se traduit néanmoins par une complexité accrue et il est plus difficile de réutiliser du code dans un autre projet Flex. Pour faciliter la réutilisation de code ActionScript dans de futurs projets Flex, stockez-le dans des fichiers ActionScript externes.

Remarque : *vous pouvez spécifier un paramètre source pour une balise `<fx:Script>`, ce qui vous permet « d'importer » du code ActionScript à partir d'un fichier externe comme s'il avait été tapé directement dans la balise `<fx:Script>`. Cependant, le fichier source que vous utilisez pour ce faire ne peut définir sa propre classe, ce qui limite les possibilités de réutilisation.*

Stockage du code dans des fichiers ActionScript

Si votre projet implique une quantité importante de code ActionScript, la meilleure solution consiste à stocker le code dans des fichiers source ActionScript (des fichiers texte dotés de l'extension .as). Un fichier ActionScript peut suivre deux structures, selon l'utilisation que vous prévoyez d'en faire dans votre application.

- Code ActionScript non structuré : les lignes de code ActionScript, y compris les instructions et les définitions de fonction, sont écrites comme si elles étaient saisies directement dans un script de scénario ou un fichier MXML.

  Rédigé de cette façon, le code est accessible par le biais de l'instruction ActionScript `include` ou de la balise `<fx:Script>` dans Flex MXML. L'instruction ActionScript `include` indique au compilateur d'insérer le contenu d'un fichier ActionScript externe à un endroit particulier d'un script et sur une étendue donnée, comme s'il avait été saisi directement. Dans le langage MXML, la balise `<fx:Script>` permet de spécifier l'attribut source qui identifie le fichier ActionScript externe à charger à cet endroit de l'application. Par exemple, la balise suivante charge un fichier ActionScript externe appelé Box.as :

  ```
  <fx:Script source="Box.as" />
  ```

- Définition d'une classe ActionScript : la définition d'une classe ActionScript ainsi que ses définitions de méthode et de propriété.

  Lorsque vous définissez une classe, vous pouvez accéder au code ActionScript correspondant en créant une occurrence de cette classe et en utilisant ses propriétés, méthodes et événements, comme vous le feriez avec toute classe ActionScript intégrée. Cela implique deux opérations :

  - Utilisez l'instruction `import` pour spécifier le nom complet de la classe, de manière à ce que le compilateur ActionScript sache où la trouver. Par exemple, pour utiliser la classe MovieClip dans ActionScript, importez cette classe à l'aide de son nom complet, en incluant le package et la classe :

    ```
    import flash.display.MovieClip;
    ```

    Une autre solution consiste à importer le package contenant la classe MovieClip, ce qui revient à écrire des instructions `import` pour chaque classe du package :

    ```
    import flash.display.*;
    ```

    Cette obligation d'importer les classes auxquelles vous faites référence dans votre code ne s'applique pas aux classes de niveau supérieur, qui ne sont pas définies dans le package.

- Rédigez du code qui fait spécifiquement référence au nom de classe. Par exemple, déclarez une variable dont le type de données est cette classe et créez une occurrence de la classe à stocker dans la variable. Lorsque vous utilisez une classe dans le code ActionScript, vous indiquez au compilateur de charger la définition de cette classe. Par exemple, si l'on considère une classe externe appelée Box, l'instruction suivante provoque la création d'une occurrence de cette classe :

  ```
  var smallBox:Box = new Box(10,20);
  ```

  Lorsque le compilateur rencontre pour la première fois la référence à la classe Box, il effectue une recherche dans le code source chargé afin de trouver la définition de cette classe.

Choix de l'outil approprié

Vous disposez de plusieurs outils (à utiliser individuellement ou en combinaison) pour l'écriture et la modification du code ActionScript.

Flash Builder

Adobe Flash Builder est le principal outil de création de projets avec la structure Flex ou de projets constitués en majorité de code ActionScript. Outre ses outils de présentation visuelle et d'édition MXML, Flash Builder comprend un éditeur ActionScript complet, qui permet de créer des projets Flex ou ActionScript seul. Flex présente de nombreux avantages, notamment un large éventail de commandes d'interface préintégrées et de commandes de disposition dynamique souples, ainsi que des mécanismes intégrés permettant de manipuler des données à distance et de lier des données externes aux éléments d'interface utilisateur. Toutefois, ces fonctions nécessitant davantage de code, les projets utilisant Flex se caractérisent par une taille de fichier SWF supérieure à celle de leurs homologues non-Flex.

Utilisez Flash Builder pour créer avec Flex, dans un seul et même outil, des applications de données sur Internet riches en fonctions, tout en modifiant du code ActionScript et MXLM et en disposant les éléments de manière visuelle.

De nombreux utilisateurs de Flash Professional qui élaborent des projets ActionScript utilisent cette application pour créer des actifs visuels et Flash Builder comme éditeur du code ActionScript.

Flash Professional

Outre ses capacités de création graphique et d'animation, Flash Professional comprend des outils qui permettent de manipuler le code ActionScript, qu'il soit joint à des éléments d'un fichier FLA ou regroupé dans des fichiers ActionScript externes. Flash Professional s'avère idéal pour les projets impliquant des animations ou vidéos conséquentes, ou lorsque vous désirez créer la plupart des actifs graphiques vous-même. Cet outil peut également vous paraître adapté au développement d'un projet ActionScript si vous préférez créer les actifs visuels et écrire le code dans une seule et même application. Flash Professional propose également des composants d'interface utilisateur préintégrés. Vous pouvez vous en servir pour réduire la taille du fichier SWF et faire appel à des outils visuels pour les envelopper dans le cadre de votre projet.

Flash Professional inclut deux outils permettant l'écriture de code ActionScript :

- Panneau Actions : disponible lorsque vous manipulez un fichier FLA, ce panneau vous permet d'écrire du code ActionScript associé aux images d'un scénario.
- Fenêtre de script : la fenêtre de script est un éditeur de texte dédié permettant de travailler sur des fichiers de code ActionScript (.as).

Editeur ActionScript tiers

Les fichiers ActionScript (.as) étant stockés comme de simples fichiers texte, tout programme susceptible de modifier des fichiers texte brut peut servir à écrire des fichiers ActionScript. Outre les produits ActionScript d'Adobe, plusieurs programmes tiers d'édition de texte ont été créés avec des fonctions propres à ActionScript. Vous pouvez écrire un fichier MXML ou des classes ActionScript à l'aide de tout éditeur de texte. Vous pouvez ensuite créer une application à l'aide du kit de développement SDK Flex. Le projet peut utiliser Flex ou consister en une application ActionScript seul. Pour certains développeurs, une autre solution consiste à écrire les classes ActionScript dans Flash Builder ou un éditeur ActionScript tiers, en combinaison avec Flash Professional pour la création du contenu graphique.

Vous pouvez choisir un éditeur ActionScript tiers dans les cas suivants :

- Vous préférez écrire le code ActionScript dans un programme distinct, tout en concevant les éléments visuels dans Flash Professional.
- Vous utilisez une application de programmation non ActionScript (par exemple pour la création de pages HTML ou l'élaboration d'application dans un autre langage de programmation) et vous souhaitez vous en servir pour le code ActionScript également.
- Vous voulez créer des projets ActionScript seul ou Flex à l'aide du kit de développement SDK Flex sans Flash Professional ou Flash Builder.

Les principaux éditeurs de code prenant en charge ActionScript sont les suivants :

- Adobe Dreamweaver® CS4
- ASDT (disponible en anglais uniquement)
- FDT (disponible en anglais uniquement)
- FlashDevelop (disponible en anglais uniquement)
- PrimalScript (disponible en anglais uniquement)
- SE|PY(disponible en anglais uniquement)
- TextMate (intégrant ActionScript et Flex)

Processus de développement ActionScript

Quelle que soit la taille de votre projet ActionScript, l'utilisation d'un processus de conception et de développement vous aidera à travailler plus efficacement. Les étapes ci-après forment le processus de développement de base pour la conception d'une application avec ActionScript 3.0 :

1 Concevez votre application.

Avant de commencer à construire votre application, décrivez-la d'une manière ou d'une autre.

2 Composez votre code ActionScript 3.0.

Vous pouvez créer du code ActionScript dans Flash Professional, Flash Builder, Dreamweaver ou un éditeur de texte.

3 Créez un projet Flash ou Flex pour exécuter votre code.

Dans Flash Professional, créez un fichier FLA, définissez les paramètres de publication, ajoutez des composants d'interface utilisateur à l'application et référencez le code ActionScript. Dans Flex, définissez l'application et ajoutez des composants d'interface utilisateur à l'aide de MXML, puis référencez le code ActionScript.

4 Publiez et testez l'application ActionScript.

A cet effet, exécutez-la au sein de l'environnement de développement et vérifiez qu'elle effectue toutes les opérations voulues.

Il n'est pas indispensable de suivre ces étapes dans cet ordre ou d'achever l'une d'elles avant de passer à la suivante. Par exemple, vous pouvez concevoir un écran de l'application (étape 1), puis créer les graphiques, boutons, etc. (étape 3), avant d'écrire le code ActionScript (étape 2) et de le tester (étape 4). Vous pouvez tout aussi bien concevoir une partie de l'écran, puis ajouter un bouton ou un élément d'interface à la fois, écrire le code ActionScript correspondant et le tester dès qu'il est prêt. Bien qu'il soit judicieux de garder à l'esprit ces quatre stades du processus de développement, il est en pratique plus efficace d'aller et venir entre ces étapes en fonction des besoins.

Création de vos propres classes

Le processus de création des classes destinées à vos projets peut paraître rébarbatif. Cependant, la partie la plus difficile de la création d'une classe est la conception de ses méthodes, propriétés et événements.

Stratégies de conception d'une classe

La conception orientée objet est un sujet complexe ; des carrières entières ont été consacrées à l'étude académique et à la pratique professionnelle de cette discipline. Voici tout de même quelques suggestions d'approches qui vous aideront à lancer votre projet.

1. Réfléchissez au rôle que jouent les occurrences de la classe dans l'application. En règle générale, les objets servent l'un des objectifs suivants :
   - Objet de valeur : ces objets constituent avant tout des conteneurs de données, c'est-à-dire qu'ils possèdent probablement plusieurs propriétés et peu de méthodes (parfois aucune). Il s'agit en général d'une représentation dans le code d'éléments clairement définis, tels qu'une classe Song (représentant une seule chanson) ou une classe Playlist (représentant un groupe conceptuel de chansons) dans une application musicale.
   - Objet d'affichage : ce type correspond à des objets qui s'affichent réellement à l'écran, par exemple des éléments d'interface, tels que des listes déroulantes ou des libellés d'état, des éléments graphiques tels que des créatures dans un jeu vidéo, etc.
   - Structure d'application : ces objets jouent un large éventail de rôles dans la logique ou le traitement effectué par les applications. Il s'agit par exemple d'un objet réalisant des calculs dans une simulation de biologie, un objet chargé de synchroniser les valeurs entre une commande physique et le volume de sortie d'une application musicale, un objet qui gère les règles d'un jeu vidéo ou une classe qui charge une image enregistrée dans une application de dessin.
2. Décidez de la fonctionnalité requise pour la classe. Les différentes fonctionnalités constituent souvent les méthodes de la classe.
3. Si la classe est destinée à servir d'objet de valeur, choisissez les données inclues dans les occurrences. Ces éléments peuvent facilement devenir des propriétés.
4. Puisque vous concevez la classe spécialement pour votre projet, le plus important est que vous établissiez la fonctionnalité nécessaire à votre application. Pour vous aider, vous pouvez répondre à ces questions :
   - Quel type d'informations l'application stocke, surveille et manipule-t-elle ? Vous pourrez alors identifier les objets de valeurs et les propriétés nécessaires.
   - Quels jeux d'actions doivent être exécutés, par exemple lors du premier chargement de l'application, lorsque l'utilisateur clique sur un bouton donné, lorsque la lecture d'un clip s'arrête, etc. ? Ces éléments constituent souvent des méthodes ou des propriétés, si les « actions » consistent à modifier des valeurs isolées.
   - Pour chaque action considérée, quelles informations sont nécessaires pour son exécution ? Ces éléments deviennent les paramètres de la méthode.

- Pendant le fonctionnement de l'application, quelles modifications, qui doivent être communiquées à d'autres parties de l'application, surviennent dans la classe ? Ces éléments forment en général des événements.

5 Si un objet existant est semblable à l'objet dont vous avez besoin mais qu'il lui manque certaines fonctionnalités que vous souhaitez ajouter, envisagez de créer une sous-classe (c.-à-d. une classe qui repose sur la fonctionnalité d'une classe existante et dont il n'est pas nécessaire de définir la fonctionnalité propre). Par exemple, pour créer une classe correspondant à un objet affiché à l'écran, appuyez-vous sur le comportement de l'un des objets d'affichage existants (MovieClip ou Sprite, par exemple), qui constituerait alors la *classe de base*, que votre classe viendrait étendre.

Ecriture du code d'une classe

Une fois que vous avez conçu votre classe ou au moins identifié les informations qu'elle manipulera et les actions qu'elle devra effectuer, l'écriture et la syntaxe à utiliser sont relativement simples.

Voici la procédure minimale de création d'une classe ActionScript :

1 Ouvrez un nouveau document texte dans un éditeur de texte ActionScript.

2 Saisissez une instruction `class` afin de définir le nom de la classe. Pour ce faire, entrez les mots `public class` puis le nom de la classe, suivi d'une paire d'accolades qui entoureront le contenu de la classe (les définitions de méthode et de propriété). Exemple :

```
public class MyClass
{
}
```

Le mot `public` indique que la classe est accessible par tout autre code. Pour d'autres possibilités, voir Attributs d'espace de noms de contrôle d'accès.

3 Entrez une instruction `package` pour indiquer le nom du package contenant votre classe. La syntaxe est le mot `package`, suivi du nom complet du package, puis d'une paire d'accolades (qui entourent l'élément structurel `class`). Par exemple, modifiez le code précédent comme suit :

```
package mypackage
{
    public class MyClass
    {
    }
}
```

4 Définissez chaque propriété de la classe à l'aide de l'instruction `var` ajoutée dans le corps de la classe. Utilisez la même syntaxe que pour la déclaration d'une variable (en y ajoutant le qualificatif `public`). Par exemple, les lignes suivantes ajoutées entre les accolades de la définition de classe permettent de créer des propriétés appelées `textProperty`, `numericProperty` et `dateProperty` :

```
public var textProperty:String = "some default value";
public var numericProperty:Number = 17;
public var dateProperty:Date;
```

5 Définissez chaque méthode de la classe à l'aide de la syntaxe utilisée pour définir une fonction. Exemple :

- Pour créer la méthode `myMethod()`, saisissez :

```
public function myMethod(param1:String, param2:Number):void
{
    // do something with parameters
}
```

- Pour créer un constructeur (une méthode spéciale appelée pendant la création d'une occurrence de classe), créez une méthode dont le nom correspond exactement au nom de la classe :

```
public function MyClass()
{
    // do stuff to set initial values for properties
    // and otherwise set up the object
    textVariable = "Hello there!";
    dateVariable = new Date(2001, 5, 11);
}
```

  Si vous n'incluez pas de méthode constructeur dans votre classe, le compilateur crée automatiquement un constructeur vide (sans paramètres ni instructions) dans la classe.

Il est possible de définir d'autres éléments de classe, mais avec plus d'implication.

- Les *accesseurs* constituent un croisement spécial entre une méthode et une propriété. Lorsque vous rédigez le code de définition d'une classe, l'accesseur s'écrit comme une méthode. Vous pouvez ainsi accomplir plusieurs actions, et pas seulement la lecture ou l'attribution d'une valeur, seules actions disponibles lors de la définition d'une propriété. Toutefois, lorsque vous créez une occurrence de la classe, vous traitez l'accesseur comme une propriété et utilisez son nom pour lire ou attribuer la valeur.
- Dans ActionScript, les événements ne sont pas définis à l'aide d'une syntaxe spécifique. Pour définir les événements de la classe, vous utilisez la fonctionnalité de la classe EventDispatcher.

Voir aussi

Gestion des événements

Exemple : Création d'une application de base

ActionScript 3.0 peut s'utiliser dans divers environnements de développement d'application, notamment les outils Flash Professional et Flash Builder, ou tout éditeur de texte.

Cet exemple décrit les différentes étapes de la création et de l'amélioration d'une application ActionScript 3.0 simple à l'aide de Flash Professional ou de Flash Builder. L'application à élaborer présente une manière simple d'utiliser les fichiers de classe externes ActionScript 3.0 dans Flash Professional et Flex.

Conception d'une application ActionScript

Cet exemple d'application ActionScript est une application standard de salutation, du type « Hello World». Sa conception est donc très simple :

- L'application se nomme HelloWorld.
- Elle affiche un seul champ de texte contenant les mots « Hello World! ».
- Elle utilise une seule classe orientée objet, appelée Greeter, qui, de par sa conception, pourra être exploitée dans un projet Flash Professional ou Flex.
- Une fois la version de base de l'application créée, vous ajouterez d'autres fonctionnalités, pour inviter l'utilisateur à saisir son nom et comparer ce nom à une liste d'utilisateurs reconnus.

Cette définition bien établie, vous pouvez commencer à créer l'application elle-même.

Création du projet HelloWorld et de la classe Greeter

Selon les décisions de conception, le code de l'application HelloWorld doit être facilement réutilisable. Pour satisfaire à cette exigence, l'application utilise une seule classe orientée objet, appelée Greeter, qui est exploitée au sein d'une application que vous créez dans Flash Builder ou Flash Professional.

Pour créer le projet HelloWorld et la classe Greeter dans Flex :

1. Dans Flash Builder, sélectionnez File > New > Flex Project.
2. Attribuez le nom Hello World au projet. Vérifiez que le type d'application correspond à « Web (runs in Adobe Flash Player) », puis cliquez sur Finish.

   Flash Builder crée le projet et l'affiche dans l'Explorateur de packages. Par défaut, le projet contient un fichier nommé HelloWorld.mxml, qui est ouvert dans l'éditeur.
3. Pour créer un fichier de classe ActionScript personnalisé dans Flash Builder, sélectionnez File > New > ActionScript Class.
4. Dans le champ Name de la boîte de dialogue New ActionScript Class, saisissez le nom de classe **Greeter**, puis cliquez sur Finish.

   Une nouvelle fenêtre de modification ActionScript s'affiche.

   Passez maintenant à la section Ajout de code à la classe Greeter.

Pour créer la classe Greeter dans Flash Professional :

1. Dans Flash Professional, sélectionnez Fichier > Nouveau.
2. Dans la boîte de dialogue Nouveau document, sélectionnez un fichier ActionScript et cliquez sur OK.

   Une nouvelle fenêtre de modification ActionScript s'affiche.
3. Choisissez Fichier > Enregistrer. Sélectionnez un dossier pour votre application, nommez le fichier ActionScript **Greeter.as**, puis cliquez sur OK.

   Passez maintenant à la section Ajout de code à la classe Greeter.

Ajout de code à la classe Greeter

La classe Greeter définit un objet, `Greeter`, que vous utilisez dans l'application HelloWorld.

Pour ajouter du code à la classe Greeter :

1. Saisissez le code suivant (susceptible d'avoir été partiellement entré à votre intention) dans le nouveau fichier :

```
package
{
    public class Greeter
    {
        public function sayHello():String
        {
            var greeting:String;
            greeting = "Hello World!";
            return greeting;
        }
    }
}
```

   La classe Greeter comporte une méthode unique, `sayHello()`, qui renvoie la chaîne « Hello World! ».

2 Sélectionnez Fichier > Enregistrer pour enregistrer ce fichier ActionScript.

La classe Greeter peut maintenant être utilisée dans une application.

Création d'une application utilisant votre code ActionScript

La classe Greeter que vous avez créée définit un ensemble autonome de fonctions logicielles, mais ne constitue pas pour autant une application complète. Pour l'utiliser, vous créez un projet Flash Professional ou Flex.

Le code requiert une occurrence de la classe Greeter. La procédure d'association de la classe Greeter à l'application est la suivante.

Pour créer une application ActionScript à l'aide de Flash Professional :

1 Choisissez Fichier > Nouveau.

2 Dans la boîte de dialogue document, sélectionnez Fichier Flash (ActionScript 3.0) et cliquez sur OK.

Une nouvelle fenêtre de document s'affiche.

3 Choisissez Fichier > Enregistrer. Sélectionnez le même dossier qui contient le fichier de classe Greeter.as, nommez le document Flash **HelloWorld.fla**, puis cliquez sur OK.

4 Dans la palette des outils de Flash Professional, sélectionnez l'outil Texte et faites-le glisser sur la scène pour définir un nouveau champ de texte d'environ 300 pixels de largeur et 100 pixels de hauteur.

5 Le champ de texte étant sélectionné sur la scène, dans le panneau Propriétés, définissez le type de texte sur « Texte dynamique » et tapez **mainText** comme nom d'occurrence du champ de texte.

6 Cliquez sur la première image du scénario principal. Sélectionnez Fenêtre > Actions pour ouvrir le panneau Actions.

7 Dans le panneau Actions, tapez le script suivant :

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello();
```

8 Enregistrez le fichier.

Passez maintenant à la section Publication et test de votre application ActionScript.

Pour créer une application ActionScript à l'aide de Flash Builder :

1 Ouvrez le fichier HelloWorld.mxml et ajoutez le code illustré ci-dessous :

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400"/>

</s:Application>
```

Le projet Flex comprend quatre balises MXML :

- Une balise `<s:Application>`, qui définit le conteneur Application
- Une balise `<s:layout>`, qui définit le style de formatage (formatage vertical) de la balise Application
- Une balise `<fx:Script>`, qui contient du code ActionScript
- Une balise `<s:TextArea>`, qui définit un champ d'affichage des messages texte destinés à l'utilisateur

Le code qui figure dans la balise `<fx:Script>` définit une méthode `initApp()` qui est appelée au chargement de l'application. La méthode `initApp()` définit la valeur du texte de `mainTxt` TextArea sur la chaîne « Hello World! » renvoyée par la méthode `sayHello()` de la classe personnalisée Greeter, que vous venez de créer.

2 Sélectionnez Fichier > Enregistrer pour enregistrer le fichier.

Passez maintenant à la section Publication et test de votre application ActionScript.

Publication et test de votre application ActionScript

Le développement logiciel est un processus itératif. Vous écrivez du code, essayez de le compiler, puis modifiez ce code jusqu'à ce que la compilation soit réussie. Vous exécutez, puis testez l'application compilée, pour voir si elle répond aux intentions de conception. Si ce n'est pas le cas, vous modifiez à nouveau le code, jusqu'à obtenir satisfaction. Les environnements de développement Flash Professional et Flash Builder permettent de publier, tester et déboguer les applications de plusieurs manières.

Voici une liste de base des étapes de test de l'application HelloWorld dans chacun de ces environnements.

Pour publier et tester une application ActionScript à l'aide de Flash Professional :

1. Publiez l'application et recherchez les erreurs de compilation. Dans Flash Professional, sélectionnez Contrôle > Tester l'animation pour compiler le code ActionScript et exécuter l'application HelloWorld.
2. Si des erreurs ou des avertissements s'affichent dans la fenêtre Sortie lorsque vous testez l' application, résolvez-les dans les fichiers HelloWorld.fla ou HelloWorld.as, puis réessayez de tester l'application.
3. S'il n'existe aucune erreur de compilation, une fenêtre Flash Player affiche l'application HelloWorld.

Vous avez créé une application orientée objet, simple mais complète, qui utilise ActionScript 3.0. Passez maintenant à la section Amélioration de l'application HelloWorld.

Pour publier et tester une application ActionScript à l'aide de Flash Builder :

1. Sélectionnez Run > Run HelloWorld.
2. L'application HelloWorld démarre.
   - Si des erreurs ou des avertissements s'affichent dans la fenêtre de sortie lorsque vous testez l' application, résolvez-les dans les fichiers HelloWorld.mxml ou Greeter.as, puis réessayez de tester l'application.
   - En l'absence d'erreurs de compilation, une fenêtre de navigateur contenant l'application HelloWorld apparaît. Elle devrait afficher le texte "Hello World!"

   Vous avez créé une application orientée objet, simple mais complète, qui utilise ActionScript 3.0. Passez maintenant à la section Amélioration de l'application HelloWorld.

Amélioration de l'application HelloWorld

Pour rendre l'application un peu plus attrayante, vous allez maintenant faire en sorte qu'elle demande le nom de l'utilisateur et le compare à une liste prédéfinie de noms.

Tout d'abord, vous devez mettre à jour la classe Greeter de manière à ajouter cette fonctionnalité. Vous devez ensuite mettre à jour l'application afin d'exploiter cette fonctionnalité.

Pour mettre à jour le fichier Greeter.as :

1. Ouvrez le fichier Greeter.as.
2. Remplacez son contenu par le suivant (les lignes nouvelles et modifiées sont signalées en gras) :

```
package
{
    public class Greeter
    {
        /**
         * Defines the names that receive a proper greeting.
         */
        public static var validNames:Array = ["Sammy", "Frank", "Dean"];

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                            + "the Enter key.";
            }
            else if (validName(userName))
            {
                greeting = "Hello, " + userName + ".";
            }
            else
            {
                greeting = "Sorry " + userName + ", you are not on the list.";
            }
            return greeting;
        }

        /**
         * Checks whether a name is in the validNames list.
         */
        public static function validName(inputName:String = ""):Boolean
        {
            if (validNames.indexOf(inputName) > -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

La classe Greeter présente maintenant plusieurs fonctions nouvelles :

- Le tableau `validNames` répertorie les noms d'utilisateur valables. Lors du chargement de la classe Greeter, ce tableau contient trois noms.

- La méthode `sayHello()` accepte désormais un nom d'utilisateur et modifie la salutation en fonction de certaines conditions. Si le nom d'utilisateur `userName` est une chaîne vide (`""`), la propriété `greeting` permet de demander le nom de l'utilisateur. Si le nom d'utilisateur est valable, la salutation devient `"Hello, `*userName*`"`. Enfin, si l'une de ces deux conditions n'est pas satisfaite, la variable `greeting` renvoie la valeur `"Sorry, `*userName*`, you are not on the list."` (Désolé, nom d'utilisateur, vous n'êtes pas sur la liste).
- La méthode `validName()` renvoie la valeur `true` si le nom entré `inputName` figure dans le tableau `validNames`, et la valeur `false` s'il ne s'y trouve pas. L'instruction `validNames.indexOf(inputName)` compare toutes les chaînes du tableau `validNames` à la chaîne `inputName`. La méthode `Array.indexOf()` renvoie la position d'index de la première occurrence d'un objet dans un tableau ou la valeur -1 si l'objet ne s'y trouve pas.

Ensuite, vous devez modifier le fichier d'application qui référence cette classe ActionScript.

Pour modifier l'application à l'aide de Flash Professional :

1. Ouvrez le fichier HelloWorld.fla.
2. Modifiez le script dans l'image 1 de façon à ce qu'une chaîne vide (`""`) soit transmise à la méthode `sayHello()` de la classe Greeter :

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```

3. Sélectionnez l'outil Texte dans la palette des outils, puis créez deux champs de texte sur la scène, l'un à côté de l'autre, et directement sous le champ de texte `mainText` existant.
4. Dans le premier nouveau champ de texte, tapez le texte **User Name:** qui servira d'étiquette.
5. Sélectionnez l'autre nouveau champ de texte et, dans l'Inspecteur des propriétés, sélectionnez Saisie de texte comme type de champ de texte. Sélectionnez le type de ligne Une seule ligne. Tapez **textIn** comme nom d'occurrence.
6. Cliquez sur la première image du scénario principal.
7. Dans le panneau Actions, ajoutez les lignes suivantes à la fin du script existant :

```
mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

Le nouveau code ajoute la fonctionnalité suivante :

- Les deux premières lignes définissent les bordures de deux champs de texte.
- Un champ de texte d'entrée tel que le champ `textIn` a un ensemble d'événements qu'il peut envoyer. La méthode `addEventListener()` vous permet de définir une fonction exécutée lorsqu'un type d'événement se produit. Dans ce cas, cet événement est le fait d'appuyer sur une touche du clavier.
- La fonction personnalisée `keyPressed()` vérifie si la touche actionnée est la touche Entrée. Si tel est le cas, elle appelle la méthode `sayHello()` de l'objet `myGreeter`, en transmettant le texte du champ de texte `textIn` comme paramètre. Cette méthode renvoie une chaîne greeting en fonction de la valeur transmise. La chaîne renvoyée est ensuite affectée à la propriété `text` du champ de texte `mainText`.

Le script complet pour l'image 1 est le suivant :

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");

mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

8 Enregistrez le fichier.

9 Choisissez Contrôle > Tester l'animation pour exécuter l'application.

Lorsque vous exécutez l'application, il vous est demandé d'entrer un nom d'utilisateur. S'il est valide (Sammy, Frank, ou Dean), l'application affiche le message de confirmation « hello ».

Pour modifier l'application à l'aide de Flash Builder :

1 Ouvrez le fichier HelloWorld.mxml.

2 Modifiez ensuite la balise `<mx:TextArea>` pour indiquer à l'utilisateur qu'elle est utilisée à des fins d'affichage uniquement. Choisissez un arrière-plan gris clair et définissez l'attribut `editable`sur `false` :

```
<s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false" />
```

3 Ajoutez à présent les lignes suivantes juste après la balise de fin de `<s:TextArea>`. Ces lignes créent un composant TextInput qui permet à l'utilisateur de saisir un nom d'utilisateur :

```
    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>
```

L'attribut `enter` définit l'action exécutée lorsque l'utilisateur appuie sur la touche Entrée dans le champ `userNameTxt`. Dans cet exemple, le code transmet le texte du champ à la méthode `Greeter.sayHello()`. Le contenu du champ `mainTxt` est modifié en conséquence.

Le fichier HelloWorld.mxml se présente comme suit :

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false"/>

    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>

</s:Application>
```

4 Enregistrez le fichier HelloWorld.mxml modifié. Sélectionnez Run > Run HelloWorld pour exécuter l'application.

Lorsque vous exécutez l'application, il vous est demandé d'entrer un nom d'utilisateur. S'il est valide (Sammy, Frank, ou Dean), l'application affiche le message de confirmation « `Hello, ` *userName* ».

Chapitre 3 : Syntaxe et langage ActionScript

ActionScript 3.0 comprend le langage ActionScript de base et l'interface de programmation d'applications (API) de la plate-forme Adobe Flash. Le langage de base correspond à la partie d'ActionScript qui définit la syntaxe du langage, ainsi que les types de données de plus haut niveau. ActionScript 3.0 fournit un accès par programmation aux moteurs d'exécution de la plate-forme Adobe Flash : Adobe Flash Player et Adobe AIR.

Présentation du langage

Les objets sont au centre du langage ActionScript 3.0 : ils constituent ses éléments fondamentaux. Chaque variable que vous déclarez, chaque fonction que vous écrivez et chaque occurrence de classe que vous créez est un objet. Un programme ActionScript 3.0 peut être comparé à un groupe d'objets qui effectuent des tâches, répondent à des événements et communiquent entre eux.

Les programmeurs qui connaissent la programmation orientée objets en langage Java ou C++ assimilent peut-être les objets à des modules contenant deux sortes de membres : les données stockées dans des variables ou des propriétés de membres et le comportement accessible par le biais de méthodes. ActionScript 3.0 définit les objets de manière similaire, mais légèrement différente. Dans ActionScript 3.0, les objets ne sont que des collections de propriétés. Ces propriétés sont des conteneurs pouvant contenir non seulement des données, mais également des fonctions ou d'autres objets. Si une fonction est associée à un objet de cette façon, il s'agit d'une méthode.

Alors qu'en théorie, la définition d'ActionScript 3.0 peut paraître un peu étrange aux programmeurs ayant l'habitude d'utiliser le langage Java ou C++, en pratique, la définition de types d'objets avec des classes ActionScript 3.0 est très semblable à la façon dont les classes sont définies dans Java ou C++. Il est important de distinguer les deux définitions d'objet lors de la présentation du modèle d'objet ActionScript et d'autres rubriques plus techniques, mais généralement, le terme *propriétés* fait référence à des variables de membre de classe, par opposition aux méthodes. Dans le manuel Guide de référence ActionScript 3.0 pour la plate-forme Adobe Flash, par exemple, le terme *propriétés* signifie variables ou propriétés de lecture et de définition. Il utilise le terme *méthodes* pour des fonctions faisant partie d'une classe.

Une légère différence entre les classes dans ActionScript et les classes dans Java ou C++ réside dans le fait que dans ActionScript, elles ne sont pas que des entités abstraites. Les classes ActionScript sont représentées par des *objets de classe* qui stockent les méthodes et les propriétés de la classe. Ainsi, des techniques pouvant sembler étranges pour les programmeurs de Java et C++ comme l'insertion d'instructions ou de code exécutable au niveau supérieur d'une classe ou d'un package sont autorisées.

Une autre différence entre les classes ActionScript et les classes Java ou C++ réside dans le fait que chaque classe ActionScript possède un *objet prototype*. Dans les versions précédentes d'ActionScript, les objets prototypes, liés les uns aux autres en *chaînes de prototypes,* servaient de base à l'ensemble de la hiérarchie d'héritage de classe. Dans ActionScript 3.0, néanmoins, les objets prototypes ne jouent qu'un rôle mineur dans le système d'héritage. Cependant, l'objet prototype peut toujours être utile comme solution de rechange aux méthodes et aux propriétés statiques si vous souhaitez partager une propriété et sa valeur parmi toutes les occurrences d'une classe.

Auparavant, les programmeurs ActionScript expérimentés pouvaient manipuler directement la chaîne de prototypes avec des éléments de langage intégrés spéciaux. Maintenant que le langage fournit une implémentation plus avancée d'une interface de programmation basée sur des classes, un grand nombre de ces éléments de langage spéciaux (`__proto__` et `__resolve,`, par exemple) ne font plus partie du langage. De plus, les optimisations du mécanisme d'héritage interne qui améliorent nettement les performances empêchent d'accéder directement au mécanisme.

Objets et classes

Dans ActionScript 3.0, chaque objet est défini par une classe. Une classe peut être considérée comme un modèle pour un type d'objet. Les définitions de classe peuvent inclure des variables et des constantes, qui comprennent des valeurs de données, et des méthodes, qui sont des fonctions encapsulant le comportement lié à la classe. Les valeurs stockées dans les propriétés peuvent être des *valeurs primitives* ou d'autres objets. Les valeurs primitives sont des nombres, des chaînes ou des valeurs booléennes.

ActionScript contient de nombreuses classes intégrées faisant partie du langage de base. Certaines de ces classes intégrées (Number, Boolean et String, par exemple), représentent les valeurs primitives disponibles dans ActionScript. D'autres, telles que les classe Array, Math et XML, définissent des objets plus complexes.

Toutes les classes, qu'elles soient intégrées ou définies par l'utilisateur, dérivent de la classe Object. Pour les programmeurs ayant une expérience avec ActionScript, il est important de noter que le type de données Object n'est plus le type de données par défaut, même si toutes les autres classes en dérivent. Dans ActionScript 2.0, les deux lignes de code suivantes étaient équivalentes car l'absence d'une annotation de type signifiait qu'une variable aurait été de type Object :

```
var someObj:Object;
var someObj;
```

ActionScript 3.0, néanmoins, présente le concept de variables non typées, qui peuvent être désignées des deux façons suivantes :

```
var someObj:*;
var someObj;
```

Une variable non typée est différente d'une variable de type Object. La différence majeure est que les variables non typées peuvent contenir la valeur spéciale `undefined`, alors qu'une variable de type Object ne le peut pas.

Vous pouvez définir vos propres classes à l'aide du mot-clé `class`. Vous disposez de trois façons différentes pour déclarer les propriétés d'une classe : vous pouvez définir des constantes avec le mot-clé `const`, des variables avec le mot-clé `var` et des propriétés de lecture et de définition au moyen des attributs `get` et `set` dans une déclaration de méthode. Vous pouvez déclarer des méthodes avec le mot-clé `function`.

Vous créez une occurrence d'une classe à l'aide de l'opérateur `new`. L'exemple suivant crée une occurrence de la classe Date appelée `myBirthday`.

```
var myBirthday:Date = new Date();
```

Packages et espaces de noms

Les packages et les espaces de noms sont des concepts associés. Les packages vous permettent de regrouper des définitions de classe de façon à faciliter le partage de code et à réduire les conflits de noms. Les espaces de noms vous permettent de contrôler la visibilité des identifiants (noms de méthode et de propriété, par exemple) et peuvent être appliqués à un code se trouvant à l'intérieur ou à l'extérieur d'un package. Utilisez des packages pour organiser vos fichiers de classe et des espaces de noms pour gérer la visibilité des méthodes et des propriétés individuelles.

Packages

Les packages dans ActionScript 3.0 sont implémentés avec des espaces de noms, mais ils ne sont pas synonymes. Lorsque vous déclarez un package, vous créez implicitement un type d'espace de noms spécial qui est sûr d'être connu lors de la compilation. Les espaces de noms, lorsque vous les créez explicitement, ne sont pas nécessairement connus au moment de la compilation.

L'exemple suivant utilise la directive `package` pour créer un simple package contenant une classe :

```
package samples
{
    public class SampleCode
    {
        public var sampleGreeting:String;
        public function sampleFunction()
        {
            trace(sampleGreeting + " from sampleFunction()");
        }
    }
}
```

Le nom de la classe dans cet exemple est SampleCode. Etant donné que la classe se trouve à l'intérieur du package samples, le compilateur qualifie automatiquement le nom de classe lors de la compilation sous la forme de son nom qualifié : samples.SampleCode. Le compilateur qualifie également les noms des propriétés ou des méthodes, de sorte que `sampleGreeting` et `sampleFunction()` deviennent respectivement `samples.SampleCode.sampleGreeting` et `samples.SampleCode.sampleFunction()`.

Un grand nombre de développeurs (notamment ceux ayant une expérience de programmation Java) placent uniquement des classes au niveau supérieur d'un package. ActionScript 3.0, cependant, prend non seulement en charge des classes au niveau supérieur d'un package, mais également des variables, des fonctions et même des instructions. Une utilisation avancée de cette fonction consiste à définir un espace de noms au niveau supérieur d'un package de façon à ce que toutes les classes contenues dans ce dernier puissent l'utiliser. Il convient néanmoins de noter que seuls deux spécificateurs d'accès, `public` et `internal`, sont autorisés au niveau supérieur d'un package. Contrairement à Java qui permet de déclarer des classes imbriquées privées, ActionScript 3.0 ne prend en charge ni les classes imbriquées, ni les classes privées.

Néanmoins, les packages ActionScript 3.0 et les packages du langage de programmation Java présentent d'autres ressemblances. Comme vous pouvez l'observer dans l'exemple précédent, les références de package entièrement qualifiées sont exprimées à l'aide de l'opérateur point (`.`), comme dans Java. Vous pouvez utiliser des packages pour organiser votre code dans une structure hiérarchique intuitive que d'autres programmeurs peuvent utiliser. Ceci facilite le partage de code en vous permettant de créer votre propre package et de le partager avec d'autres personnes, et d'utiliser des packages créés par d'autres personnes dans votre code.

L'utilisation de packages vous permet également de vérifier que les noms d'identifiant que vous utilisez sont uniques et qu'ils ne sont pas incompatibles avec d'autres noms d'identifiant. Pour certaines personnes, il s'agit d'ailleurs de l'avantage principal des packages. Par exemple, deux programmeurs qui souhaitent partager leur code peuvent chacun avoir créé une classe appelée SampleCode. Sans packages, il y aurait un conflit de noms, et la seule façon de résoudre le problème serait de renommer l'une des classes. Mais avec des packages, vous pouvez éviter le conflit de noms en plaçant l'une des classes (ou de préférence les deux) dans des packages avec des noms uniques.

Vous pouvez également inclure des points intégrés dans le nom de votre package afin de créer des packages imbriqués. Ceci vous permet de créer une organisation hiérarchique des packages. Le package flash.display, fourni par ActionScript 3.0, en est un bon exemple. Il est imbriqué au sein du package flash.

La majeure partie d'ActionScript 3.0 se trouve dans le package flash. Par exemple, le package flash.display contient l'API de la liste d'affichage, et le package flash.events contient le nouveau modèle d'événement.

Création de packages

ActionScript 3.0 vous permet d'organiser vos packages, classes et fichiers source avec une grande souplesse. Les versions précédentes d'ActionScript autorisaient uniquement une classe par fichier source et exigeaient que le nom du fichier source corresponde au nom de la classe. ActionScript 3.0 vous permet d'inclure plusieurs classes dans un fichier source, mais une seule classe dans chaque fichier peut être utilisée par le code externe à ce fichier. En d'autres termes, une seule classe dans chaque fichier peut être déclarée à l'intérieur d'une déclaration de package. Vous devez déclarer toute classe supplémentaire à l'extérieur de votre définition de package. Elles sont alors invisibles pour le code externe à ce fichier source. Le nom de la classe déclarée à l'intérieur de la définition de package doit correspondre au nom du fichier source.

ActionScript 3.0 permet également de déclarer des packages avec davantage de flexibilité. Dans les versions précédentes d'ActionScript, les packages représentaient simplement des répertoires dans lesquels vous placiez des fichiers source, et vous ne déclariez pas les packages avec l'instruction `package` mais incluiez plutôt le nom du package dans le nom de classe complet dans votre déclaration de classe. Même si les packages continuent à représenter des répertoires dans ActionScript 3.0, leur contenu n'est pas limité aux seules classes. Dans ActionScript 3.0, vous utilisez l'instruction `package` pour déclarer un package, ce qui signifie que vous pouvez également déclarer des variables, des fonctions et des espaces de noms au niveau supérieur d'un package. Vous pouvez également y inclure des instructions exécutables. Si vous déclarez des variables, des fonctions ou des espaces de noms à ce niveau, les seuls attributs disponibles sont `public` et `internal`, et une seule déclaration au niveau du package par fichier peut utiliser l'attribut `public`, que cette déclaration soit une classe, une variable, une fonction ou un espace de noms.

Les packages sont utiles pour organiser votre code et éviter les conflits de noms. Vous ne devez pas confondre le concept de packages avec le concept distinct d'héritage de classe. Deux classes se trouvant dans un même package ont un espace de noms en commun mais elles ne sont pas nécessairement liées l'une à l'autre. De même, il se peut qu'un package imbriqué n'ait aucun lien sémantique avec son package parent.

Importation de packages

Si vous souhaitez utiliser une classe se trouvant à l'intérieur d'un package, vous devez importer soit le package soit la classe en question. Ceci diffère d'ActionScript 2.0 où l'importation de classes était facultative.

Par exemple, revenons à l'exemple de classe SampleCode présenté précédemment. Si la classe se trouve dans un package appelé samples, vous devez utiliser l'une des instructions d'importation suivantes avant d'utiliser la classe SampleCode :

```
import samples.*;
```

ou

```
import samples.SampleCode;
```

En général, les instructions `import` doivent être aussi spécifiques que possible. Si vous envisagez d'utiliser uniquement la classe SampleCode issue du package samples, importez uniquement la classe SampleCode au lieu du package entier auquel elle appartient. L'importation des packages entiers peut provoquer des conflits de noms inattendus.

Vous devez également placer le code source qui définit le package ou la classe dans votre *chemin de classe*. Le chemin de classe est une liste définie par l'utilisateur de chemins de répertoire locaux qui détermine l'endroit où le compilateur recherche des classes et des packages importés. Le chemin de classe est parfois appelé *chemin de création* ou *chemin source*.

Une fois que vous avez importé correctement la classe ou le package, vous pouvez utiliser le nom entièrement qualifié de la classe (samples.SampleCode) ou simplement le nom de la classe (SampleCode).

Les noms entièrement qualifiés sont utiles lorsque des classes, des méthodes ou des propriétés ayant le même nom génèrent un code ambigu mais ils peuvent être difficiles à gérer si vous les utilisez pour tous les identifiants. Par exemple, l'utilisation du nom entièrement qualifié génère un code détaillé lorsque vous instanciez une occurrence de classe SampleCode :

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

Plus les niveaux de packages imbriqués augmentent, moins votre code est lisible. Lorsque vous pensez que des identifiants ambigus ne sont pas un problème, vous pouvez rendre votre code plus lisible en utilisant des identifiants simples. Par exemple, l'instanciation d'une nouvelle occurrence de la classe SampleCode est beaucoup moins longue si vous utilisez uniquement l'identifiant de classe :

```
var mySample:SampleCode = new SampleCode();
```

Si vous tentez d'utiliser des noms d'identifiant sans importer au préalable la classe ou le package approprié, il est impossible pour le compilateur de trouver les définitions de classe. En revanche, si vous importez un package ou une classe, toute tentative de définition d'un nom qui provoque un conflit avec un nom importé génère une erreur.

Lors de la création d'un package, le spécificateur d'accès par défaut pour tous les membres de ce package est `internal`, ce qui signifie que, par défaut, les membres du package ne sont visibles que pour d'autres membres de ce package. Si vous souhaitez qu'une classe soit disponible pour un code externe au package, vous devez la déclarer `public`. Par exemple, le package suivant contient deux classes, SampleCode et CodeFormatter :

```
// SampleCode.as file
package samples
{
    public class SampleCode {}
}

// CodeFormatter.as file
package samples
{
    class CodeFormatter {}
}
```

La classe SampleCode est visible en dehors du package car elle est déclarée comme classe `public`. La classe CodeFormatter, cependant, est visible uniquement dans le package samples. Toute tentative d'accès à la classe CodeFormatter en dehors du package samples donne lieu à une erreur, comme l'indique l'exemple suivant :

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // okay, public class
var myFormatter:CodeFormatter = new CodeFormatter(); // error
```

Si vous souhaitez que les deux classes soient disponibles en dehors du package, vous devez les déclarer `public`. Vous ne pouvez pas appliquer l'attribut `public` à la déclaration de package.

Les noms entièrement qualifiés sont utiles pour résoudre les conflits de noms pouvant se produire lors de l'utilisation de packages. Un tel scénario peut se produire si vous importez deux packages qui définissent des classes ayant le même identifiant. Par exemple, considérons le package suivant, qui possède également une classe appelée SampleCode :

```
package langref.samples
{
    public class SampleCode {}
}
```

Si vous importez les deux classes, comme suit, vous avez un conflit de noms lorsque vous utilisez la classe SampleCode :

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // name conflict
```

Le compilateur n'a aucun moyen de savoir quelle classe SampleCode il doit utiliser. Pour résoudre ce conflit, vous devez utiliser le nom entièrement qualifié de chaque classe, comme suit :

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

Remarque : *les programmeurs ayant une expérience C++ confondent souvent l'instruction `import` avec l'instruction `#include`. La directive `#include` est nécessaire dans C++ car les compilateurs C++ traitent un fichier à la fois et ne cherchent pas de définitions de classe dans d'autres fichiers, à moins qu'un fichier d'en-tête ne soit inclus explicitement. ActionScript 3.0 a une directive `include`, mais elle n'est pas conçue pour importer des classes et des packages. Pour importer des classes ou des packages dans ActionScript 3.0, vous devez utiliser l'instruction `import` et placer le fichier source qui contient le package dans le chemin de classe.*

Espaces de noms

Les espaces de noms vous permettent de contrôler la visibilité des propriétés et des méthodes que vous créez. Considérez les spécificateurs de contrôle d'accès `public`, `private`, `protected` et `internal` comme des espaces de noms intégrés. Si ces spécificateurs de contrôle d'accès prédéfinis ne répondent pas à vos besoins, vous pouvez créer vos propres espaces de noms.

Si vous avez l'habitude d'utiliser des espaces de noms XML, cette section ne vous fournira pas de nouvelles informations, bien que la syntaxe et les détails de l'implémentation d'ActionScript soient légèrement différents de ceux d'XML. Si vous n'avez jamais travaillé avec des espaces de noms, le concept est simple, mais l'implémentation possède une terminologie particulière que vous devrez apprendre.

Pour comprendre comment fonctionnent les espaces de noms, il convient de savoir que le nom d'une propriété ou d'une méthode contient deux parties : un identifiant et un espace de noms. L'identifiant est ce que vous considérez généralement comme un nom. Par exemple, les identifiants dans la définition de classe suivante sont `sampleGreeting` et `sampleFunction()` :

```
class SampleCode
{
    var sampleGreeting:String;
    function sampleFunction () {
        trace(sampleGreeting + " from sampleFunction()");
    }
}
```

Lorsque les définitions ne sont pas précédées d'un attribut d'espace de noms, leurs noms sont qualifiés par l'espace de noms `internal` par défaut, ce qui signifie qu'ils sont visibles uniquement aux appelants du même package. Si le compilateur est défini sur le mode strict, il génère un avertissement indiquant que l'espace de noms `internal` s'applique à tout identifiant sans attribut d'espace de noms. Pour vérifier qu'un identifiant est disponible partout, vous devez spécifiquement faire précéder le nom de l'identifiant de l'attribut `public`. Dans l'exemple de code précédent, `sampleGreeting` et `sampleFunction()` ont une valeur d'espace de noms d'`internal`.

Vous devez suivre trois étapes de base lorsque vous utilisez des espaces de noms. Premièrement, vous devez définir l'espace de noms à l'aide du mot-clé `namespace`. Par exemple, le code suivant définit l'espace de noms `version1` :

```
namespace version1;
```

Deuxièmement, vous appliquez votre espace de noms en l'utilisant à la place d'un spécificateur de contrôle d'accès dans une déclaration de méthode ou de propriété. L'exemple suivant place une fonction appelée `myFunction()` dans l'espace de noms `version1` :

```
version1 function myFunction() {}
```

Troisièmement, une fois que vous avez appliqué l'espace de noms, vous pouvez le référencer à l'aide de la directive `use` ou en qualifiant le nom d'un identifiant avec un espace de nom. L'exemple suivant fait référence à la fonction `myFunction()` à l'aide de la directive `use` :

```
use namespace version1;
myFunction();
```

Vous pouvez également utiliser un nom qualifié pour référencer la fonction `myFunction()`, comme l'indique l'exemple suivant :

```
version1::myFunction();
```

Définition des espaces de noms

Les espaces de noms contiennent une valeur, l'URI (Uniform Resource Identifier), parfois appelée *nom d'espace de noms*. Un URI vous permet de vérifier que votre définition d'espace de noms est unique.

Vous créez un espace de noms en déclarant sa définition de deux façons différentes. Vous pouvez soit définir un espace de noms avec un URI explicite, comme vous définiriez un espace de noms XML, soit omettre l'URI. L'exemple suivant indique comment un espace de noms peut être défini à l'aide d'un URI :

```
namespace flash_proxy = "http://www.adobe.com/flash/proxy";
```

L'URI sert de chaîne d'identification unique pour cet espace de noms. Si vous omettez l'URI, comme dans l'exemple suivant, le compilateur crée une chaîne d'identification interne unique à la place. Vous n'avez pas accès à cette chaîne d'identification interne.

```
namespace flash_proxy;
```

Une fois que vous avez défini un espace de noms (avec ou sans URI), vous ne pouvez pas le redéfinir dans le même domaine. Une tentative de définition d'un espace de noms ayant été défini dans le même domaine génère une erreur du compilateur.

Si un espace de noms est défini dans un package ou une classe, il risque de ne pas être visible au code externe à ce package ou à cette classe, à moins que vous n'utilisiez le spécificateur de contrôle d'accès approprié. Par exemple, le code suivant indique l'espace de noms `flash_proxy` défini dans le package flash.utils. Dans l'exemple suivant, l'absence de spécificateur de contrôle d'accès signifie que l'espace de noms `flash_proxy` est visible uniquement au code dans le package flash.utils et non au code externe au package :

```
package flash.utils
{
    namespace flash_proxy;
}
```

Le code suivant utilise l'attribut `public` pour rendre l'espace de noms `flash_proxy` visible au code externe au package :

```
package flash.utils
{
    public namespace flash_proxy;
}
```

Application d'espaces de noms

Appliquer un espace de noms signifie placer une définition dans un espace de noms. Les définitions que vous pouvez placer dans des espaces de noms peuvent être des fonctions, des variables et des constantes (vous ne pouvez pas placer une classe dans un espace de noms personnalisé).

Supposez, par exemple, qu'une fonction soit déclarée à l'aide de l'espace de noms de contrôle d'accès `public`. L'utilisation de l'attribut `public` dans une définition de fonction place la fonction dans l'espace de noms public et la rend visible à tout le code. Une fois que vous avez défini un espace de noms, vous pouvez l'utiliser de la même façon que l'attribut `public`, et la définition est disponible pour le code pouvant référencer votre espace de noms personnalisé. Par exemple, si vous définissez un espace de noms `example1`, vous pouvez ajouter une méthode appelée `myFunction()` à l'aide de `example1` comme attribut, tel que l'indique l'exemple suivant :

```
namespace example1;
class someClass
{
    example1 myFunction() {}
}
```

Déclarer la méthode `myFunction()` à l'aide de l'espace de noms `example1` comme attribut signifie que la méthode appartient à l'espace de noms `example1`.

Tenez compte des points suivants lorsque vous appliquez des espaces de noms :

- Vous pouvez appliquer un seul espace de noms à chaque déclaration.
- Il n'existe aucun moyen d'appliquer un attribut d'espace de noms à plusieurs définitions simultanément. En d'autres termes, si vous souhaitez appliquer votre espace de noms à dix fonctions différentes, vous devez ajouter votre espace de noms comme attribut à chacune des dix définitions de fonction.
- Si vous appliquez un espace de noms, vous ne pouvez pas spécifier un spécificateur de contrôle d'accès car les espaces de noms et les spécificateurs de contrôle d'accès s'excluent mutuellement. En d'autres termes, vous ne pouvez pas déclarer une fonction ou une propriété comme `public`, `private`, `protected` ou `internal` si vous appliquez votre espace de noms.

Référence d'espaces de noms

Il est inutile de référencer explicitement un espace de noms lorsque vous utilisez une méthode ou une propriété déclarée avec l'un des espaces de noms de contrôle d'accès (`public`, `private`, `protected` et `internal`, par exemple). En effet, l'accès à ces espaces de noms spéciaux dépend du contexte. Par exemple, les définitions placées dans l'espace de noms `private` sont automatiquement disponibles pour le code dans la même classe. Pour les espaces de noms que vous définissez, cependant, le contexte ne compte pas. Pour utiliser une méthode ou une propriété que vous avez placée dans un espace de noms personnalisé, vous devez référencer celui-ci.

Vous pouvez référencer des espaces de noms avec la directive `use namespace` ou qualifier le nom avec l'espace de noms à l'aide du ponctuateur de qualificatif de nom (`::`). Le fait de référencer un espace de noms avec la directive `use namespace` ouvre l'espace de noms, ce qui permet de l'appliquer à n'importe quel identifiant non qualifié. Par exemple, si vous avez défini l'espace de noms `example1`, vous pouvez accéder aux noms dans cet espace de noms en utilisant `use namespace example1` :

```
use namespace example1;
myFunction();
```

Vous pouvez ouvrir plusieurs espaces de noms simultanément. Une fois que vous avez ouvert un espace de noms avec `use namespace`, il reste ouvert dans le bloc de code dans lequel il a été ouvert. Il n'existe aucun moyen pour fermer explicitement un espace de noms.

Néanmoins, le fait d'avoir plusieurs espaces de noms ouverts augmente la probabilité que des conflits de noms se produisent. Si vous préférez ne pas ouvrir d'espace de noms, vous pouvez éviter la directive `use namespace` en qualifiant le nom de la propriété ou de la méthode avec l'espace de noms et le ponctuateur de qualificatif de nom. Par exemple, le code suivant indique comment vous pouvez qualifier le nom `myFunction()` avec l'espace de noms `example1` :

```
example1::myFunction();
```

Utilisation d'espaces de noms

Vous pouvez trouver un exemple d'espace de noms, tiré du monde réel, utilisé pour éviter des conflits sur les noms dans la classe flash.utils.Proxy qui fait partie d'ActionScript 3.0. La classe Proxy, qui remplace la propriété `Object.__resolve` d'ActionScript 2.0, vous permet d'intercepter les références aux propriétés ou aux méthodes non définies avant qu'une erreur ne se produise. Toutes les méthodes de la classe Proxy se trouvent dans l'espace de noms `flash_proxy` afin d'empêcher les conflits de noms.

Pour mieux comprendre comment l'espace de noms `flash_proxy` est utilisé, vous devez savoir comment utiliser la classe Proxy. La fonctionnalité de la classe Proxy est disponible uniquement aux classes qui héritent d'elle. En d'autres termes, si vous souhaitez utiliser les méthodes de la classe Proxy d'un objet, la définition de classe de l'objet doit étendre la classe Proxy. Par exemple, si vous souhaitez intercepter des tentatives d'appel d'une méthode non définie, vous devez étendre la classe Proxy puis remplacer la méthode `callProperty()` de la classe Proxy.

L'implémentation des espaces de noms est généralement un processus en trois étapes (définition, application et référence d'un espace de noms). Etant donné que vous n'appelez jamais explicitement une méthode de la classe Proxy, cependant, l'espace de noms `flash_proxy` est défini et appliqué uniquement, jamais référencé. ActionScript 3.0 définit l'espace de noms `flash_proxy` et l'applique dans la classe Proxy. Votre code doit uniquement appliquer l'espace de noms `flash_proxy` à des classes qui étendent la classe Proxy.

L'espace de noms `flash_proxy` est défini dans le package flash.utils comme illustré ci-dessous :

```
package flash.utils
{
    public namespace flash_proxy;
}
```

L'espace de noms est appliqué aux méthodes de la classe Proxy comme indiqué dans l'extrait suivant issu de la classe Proxy :

```
public class Proxy
{
    flash_proxy function callProperty(name:*, ... rest):*
    flash_proxy function deleteProperty(name:*):Boolean
    ...
}
```

Comme l'indique le code suivant, vous devez d'abord importer la classe Proxy et l'espace de noms `flash_proxy`. Vous devez ensuite déclarer votre classe de façon à ce qu'elle étende la classe Proxy (vous devez également ajouter l'attribut `dynamic` si vous compilez en mode strict). Lorsque vous remplacez la méthode `callProperty()`, vous devez utiliser l'espace de noms `flash_proxy`.

```
package
{
    import flash.utils.Proxy;
    import flash.utils.flash_proxy;

    dynamic class MyProxy extends Proxy
    {
        flash_proxy override function callProperty(name:*, ...rest):*
        {
            trace("method call intercepted: " + name);
        }
    }
}
```

Si vous créez une occurrence de la classe MyProxy et appelez une méthode non définie (la méthode `testing()` appelée dans l'exemple suivant, par exemple), votre objet Proxy intercepte l'appel de méthode et exécute les instructions se trouvant dans la méthode `callProperty()` remplacée (dans ce cas, une instruction `trace()` simple).

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // method call intercepted: testing
```

Le fait que les méthodes de la classe Proxy se trouvent dans l'espace de noms `flash_proxy` présente deux avantages. Premièrement, le fait d'avoir un espace de noms séparé réduit l'encombrement dans l'interface publique des classes qui étendent la classe Proxy (iI existe environ douze méthodes dans la classe Proxy que vous pouvez remplacer. Elles ne sont pas conçues pour être appelées directement. Le fait de toutes les placer dans l'espace de noms public peut prêter à confusion). Deuxièmement, le fait d'utiliser l'espace de noms `flash_proxy` évite les conflits de nom si votre sous-classe Proxy contient des méthodes d'occurrence avec des noms correspondant à l'une des méthodes de la classe Proxy. Par exemple, vous pouvez nommer l'une de vos méthodes `callProperty()`. Le code suivant est acceptable car votre version de la méthode `callProperty()` se trouve dans un espace de noms différent :

```
dynamic class MyProxy extends Proxy
{
    public function callProperty() {}
    flash_proxy override function callProperty(name:*, ...rest):*
    {
        trace("method call intercepted: " + name);
    }
}
```

Vous pouvez également utiliser des espaces de noms pour accéder à des méthodes ou à des propriétés autrement qu'avec les quatre spécificateurs de contrôle d'accès (`public`, `private`, `internal` et `protected`). Par exemple, vous pouvez avoir des méthodes d'utilitaire éparpillées sur plusieurs packages. Vous souhaitez que ces méthodes soient disponibles pour tous vos packages, mais vous ne souhaitez pas qu'elles soient publiques. Pour cela, vous pouvez créer un espace de noms et l'utiliser comme spécificateur de contrôle d'accès spécial.

L'exemple suivant utilise un espace de noms défini par l'utilisateur pour regrouper deux fonctions se trouvant dans différents packages. En les regroupant dans le même espace de noms, vous pouvez rendre les deux fonctions visibles à une classe ou à un package au moyen d'une seule instruction `use namespace`.

Cet exemple utilise quatre fichiers pour démontrer la technique. Tous les fichiers doivent se trouver dans votre chemin de classe. Le premier fichier, myInternal.as, sert à définir l'espace de noms `myInternal`. Etant donné que le fichier se trouve dans un package appelé example, vous devez placer le fichier dans un dossier appelé example. L'espace de noms est marqué comme `public` pour pouvoir être importé dans d'autres packages.

```
// myInternal.as in folder example
package example
{
    public namespace myInternal = "http://www.adobe.com/2006/actionscript/examples";
}
```

Le deuxième et le troisième fichiers, Utility.as et Helper.as, définissent les classes qui contiennent des méthodes devant être disponibles pour d'autres packages. La classe Utility se trouve dans le package example.alpha, ce qui signifie que le fichier doit être placé dans un dossier appelé alpha qui est un sous-dossier du dossier example. La classe Helper se trouve dans le package example.beta, ce qui signifie que le fichier doit être placé dans un dossier appelé beta qui est également un sous-dossier du dossier example. Ces deux packages, example.alpha et example.beta, doivent importer l'espace de noms avant de l'utiliser.

```
// Utility.as in the example/alpha folder
package example.alpha
{
    import example.myInternal;

    public class Utility
    {
        private static var _taskCounter:int = 0;

        public static function someTask()
        {
            _taskCounter++;
        }

        myInternal static function get taskCounter():int
        {
            return _taskCounter;
        }
    }
}
```

```
// Helper.as in the example/beta folder
package example.beta
{
    import example.myInternal;

    public class Helper
    {
        private static var _timeStamp:Date;

        public static function someTask()
        {
            _timeStamp = new Date();
        }

        myInternal static function get lastCalled():Date
        {
            return _timeStamp;
        }
    }
}
```

Le quatrième fichier, NamespaceUseCase.as, est la classe de l'application principale et doit être un frère pour le dossier example. Dans Flash Professional, cette classe ferait office de classe de document du fichier FLA. La classe NamespaceUseCase importe également l'espace de noms `myInternal` et l'utilise pour appeler les deux méthodes statiques qui résident dans les autres packages. L'exemple utilise des méthodes statiques pour simplifier le code uniquement. Les méthodes statiques et d'occurrence peuvent être placées dans l'espace de noms `myInternal`.

```
// NamespaceUseCase.as
package
{
    import flash.display.MovieClip;
    import example.myInternal; // import namespace
    import example.alpha.Utility;// import Utility class
    import example.beta.Helper;// import Helper class

    public class NamespaceUseCase extends MovieClip
    {
        public function NamespaceUseCase()
        {
            use namespace myInternal;

            Utility.someTask();
            Utility.someTask();
            trace(Utility.taskCounter); // 2

            Helper.someTask();
            trace(Helper.lastCalled); // [time someTask() was last called]
        }
    }
}
```

Variables

Les variables vous permettent de stocker des valeurs que vous utilisez dans votre programme. Pour déclarer une variable, vous devez utiliser l'instruction `var` avec le nom de variable. Dans ActionScript 3.0, l'utilisation de l'instruction `var` est obligatoire. Par exemple, la ligne d'ActionScript suivante déclare une variable appelée `i` :

```
var i;
```

Si vous omettez l'instruction `var` lors de la déclaration d'une variable, vous obtenez une erreur de compilateur en mode strict et une erreur d'exécution en mode standard. Par exemple, la ligne de code suivante provoque une erreur si la variable `i` n'a pas été définie précédemment :

```
i; // error if i was not previously defined
```

Vous devez associer une variable à un type de données lorsque vous déclarez la variable. La déclaration d'une variable sans désigner son type est autorisée mais provoque un avertissement du compilateur en mode strict. Vous désignez le type d'une variable en ajoutant le nom de variable avec deux-points (:), suivi par le type de variable. Par exemple, le code suivant déclare une variable `i` de type int :

```
var i:int;
```

Vous pouvez affecter une valeur à une variable à l'aide de l'opérateur d'affectation (`=`). Par exemple, le code suivant déclare une variable `i` et lui affecte la valeur 20 :

```
var i:int;
i = 20;
```

Il peut être plus pratique d'affecter une valeur à une variable au moment où vous déclarez cette dernière, comme dans l'exemple suivant :

```
var i:int = 20;
```

La technique d'affectation d'une valeur à une variable au moment de sa déclaration est couramment utilisée non seulement lors de l'affectation de valeurs primitives (nombres entiers et chaînes) mais également lors de la création d'un tableau ou de l'instanciation de l'occurrence d'une classe. L'exemple suivant illustre un tableau déclaré auquel est affectée une valeur à l'aide d'une ligne de code.

```
var numArray:Array = ["zero", "one", "two"];
```

Vous pouvez créer une occurrence d'une classe à l'aide de l'opérateur `new`. L'exemple suivant crée une occurrence d'une classe appelée `CustomClass` et affecte une référence à l'occurrence de classe nouvellement créée sur la variable appelée `customItem` :

```
var customItem:CustomClass = new CustomClass();
```

Si vous avez plusieurs variables à déclarer, vous pouvez le faire sur une ligne de code à l'aide de l'opérateur virgule (`,`) pour séparer les variables. Par exemple, le code suivant déclare trois variables sur une ligne de code :

```
var a:int, b:int, c:int;
```

Vous pouvez également affecter des valeurs à chacune des variables sur une même ligne de code. Par exemple, le code suivant déclare trois variables (`a`, `b`, et `c`) et affecte une valeur à chacune d'entre elles :

```
var a:int = 10, b:int = 20, c:int = 30;
```

Bien que vous puissiez utiliser l'opérateur virgule pour regrouper des déclarations de variable dans une instruction, ceci risque de rendre votre code moins lisible.

Présentation du domaine d'une variable

Le *domaine* d'une variable est la partie de votre code dans laquelle une référence lexicale peut accéder à la variable. Une variable *globale* est une variable définie dans toutes les parties de votre code, alors qu'une variable *locale* est une variable définie dans une seule partie de votre code. Dans ActionScript 3.0, le domaine des variables est toujours limité à la fonction ou à la classe dans laquelle elles sont déclarées. Une variable globale est une variable que vous définissez en dehors de toute définition de classe ou de fonction. Par exemple, le code suivant crée une variable globale `strGlobal` en la déclarant en dehors de toute fonction. L'exemple indique qu'une variable globale est disponible à la fois à l'intérieur et à l'extérieur de la définition de fonction.

```
var strGlobal:String = "Global";
function scopeTest()
{
    trace(strGlobal); // Global
}
scopeTest();
trace(strGlobal); // Global
```

Vous déclarez une variable locale en déclarant la variable à l'intérieur d'une définition de fonction. La plus petite partie de code pour laquelle vous pouvez définir une variable locale est une définition de fonction. Une variable locale déclarée dans une fonction existe uniquement dans celle-ci. Par exemple, si vous déclarez une variable appelée `str2` dans la fonction `localScope()`, cette variable n'est pas disponible en dehors de la fonction.

```
function localScope()
{
    var strLocal:String = "local";
}
localScope();
trace(strLocal); // error because strLocal is not defined globally
```

Si le nom attribué à votre variable locale est déjà déclaré en tant que variable globale, la définition locale masque la définition globale tant que la variable locale est dans le domaine. La variable globale persiste en dehors de la fonction. Par exemple, le code suivant crée une variable globale de type chaîne appelée `str1`, puis une variable locale du même nom dans la fonction `scopeTest()`. L'instruction `trace` placée dans la fonction génère la valeur locale de la variable, tandis que celle qui est placée en dehors de la fonction génère la valeur globale de la variable.

```
var str1:String = "Global";
function scopeTest ()
{
    var str1:String = "Local";
    trace(str1); // Local
}
scopeTest();
trace(str1); // Global
```

Les variables ActionScript, contrairement aux variables dans C++ et Java, n'ont pas de domaine de niveau bloc. Un bloc de code est un groupe d'instructions placé entre une accolade d'ouverture ({) et une accolade de fermeture (}). Dans certains langages de programmation tels que C++ et Java, les variables déclarées dans un bloc de code ne sont pas disponibles en dehors de ce bloc de code. Cette restriction de domaine est appelée domaine de niveau bloc et n'existe pas dans ActionScript. Si vous déclarez une variable à l'intérieur d'un bloc de code, elle est disponible non seulement dans celui-ci mais également dans d'autres parties de la fonction à laquelle il appartient. Par exemple, la fonction suivante contient des variables définies dans différents domaines de bloc. Toutes les variables sont disponibles dans la fonction.

```
function blockTest (testArray:Array)
{
    var numElements:int = testArray.length;
    if (numElements > 0)
    {
        var elemStr:String = "Element #";
        for (var i:int = 0; i < numElements; i++)
        {
            var valueStr:String = i + ": " + testArray[i];
            trace(elemStr + valueStr);
        }
        trace(elemStr, valueStr, i); // all still defined
    }
    trace(elemStr, valueStr, i); // all defined if numElements > 0
}

blockTest(["Earth", "Moon", "Sun"]);
```

Une implication intéressante de l'absence de domaine de niveau bloc est que vous pouvez lire ou écrire dans une variable avant de la déclarer, tant qu'elle est déclarée avant que la fonction se termine. Ceci est dû à une technique appelée *hissage*, qui signifie que le compilateur déplace toutes les déclarations de variable en haut de la fonction. Par exemple, le code suivant compile même si la fonction `trace()` initiale pour la variable `num` se produit avant la déclaration de la variable `num` :

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

Néanmoins, le compilateur ne hisse aucune instruction d'affectation. Ceci explique pourquoi la fonction `trace()` initiale de `num` fournit `NaN` (pas un nombre), qui est la valeur par défaut pour des variables du type de données Number. Cela signifie que vous pouvez affecter des valeurs à des variables même avant leur déclaration, comme indiqué dans l'exemple suivant :

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

Valeurs par défaut

Une *valeur par défaut* est le contenu d'une variable avant que vous définissiez sa valeur. Vous *initialisez* une variable lorsque vous lui affectez sa première valeur. Si vous déclarez une variable sans définir sa valeur, cette variable est *non initialisée*. La valeur d'une variable non initialisée dépend de son type de données. Le tableau suivant décrit les valeurs par défaut des variables, organisées par type de données :

Type de données	Valeur par défaut
Boolean	`false`
int	0
Number	`NaN`
Object	`null`
String	`null`

Type de données	Valeur par défaut
uint	0
Non déclaré (équivalent à l'annotation de type `*`)	`undefined`
Toutes les autres classes, y compris celles définies par l'utilisateur	`null`

Pour les variables de type Number, la valeur par défaut est `NaN` (pas un nombre), qui est une valeur spéciale définie par la norme IEEE-754 pour indiquer une valeur qui ne représente pas un nombre.

Si vous déclarez une variable mais pas son type de données, le type de données par défaut `*` lui est attribué, ce qui signifie qu'elle n'est pas typée. Si vous n'initialisez pas non plus de variable non typée avec une valeur, sa valeur par défaut est `undefined`.

Pour les types de données autres que Boolean, Number, int, et uint, la valeur par défaut d'une variable non initialisée est `null`. Ceci s'applique à toutes les classes définies par ActionScript 3.0, ainsi qu'aux classes personnalisées que vous créez.

La valeur `null` n'est pas une valeur valide pour des variables de type Boolean, Number, int ou uint. Si vous tentez d'affecter la valeur `null` à une telle variable, la valeur est convertie en la valeur par défaut pour ce type de données. Pour les variables de type Object, vous pouvez affecter une valeur `null`. Si vous tentez d'affecter la valeur `undefined` à une variable de type Object, la valeur est convertie en `null`.

Pour les variables de type Number, il existe une fonction de niveau supérieur spéciale appelée `isNaN()` qui renvoie la valeur Boolean `true` si la variable n'est pas un nombre, et `false` autrement.

Types de données

Un *type de données* définit un ensemble de valeurs. Par exemple, le type de données Boolean est l'ensemble de deux valeurs exactement : `true` et `false`. Outre le type de données Boolean, ActionScript 3.0 définit plusieurs autres types de données couramment utilisés tels que String, Number et Array. Vous pouvez définir vos types de donnés en utilisant des classes ou des interfaces afin de définir un ensemble de valeurs personnalisé. Toutes les valeurs dans ActionScript 3.0, qu'elles soient primitives ou complexes, sont des objets.

Une *valeur primitive* est une valeur appartenant à l'un des types de données suivants : Boolean, int, Number, String et uint. L'utilisation de valeurs primitives est généralement plus rapide que l'utilisation de valeurs complexes car ActionScript stocke les valeurs primitives de façon à permettre des optimisations de vitesse et de mémoire.

Remarque : *pour les lecteurs intéressés par les détails techniques, ActionScript stocke les valeurs primitives en tant qu'objets inaltérables. Le fait qu'elles soient stockées en tant qu'objets inaltérables signifie que le transfert par référence est identique au transfert par valeur. Ceci réduit l'utilisation de la mémoire et augmente la vitesse d'exécution car les références sont généralement beaucoup plus petites que les valeurs elles-mêmes.*

Une *valeur complexe* est une valeur qui n'est pas une valeur primitive. Les types de données qui définissent des ensembles de valeurs complexes comprennent Array, Date, Error, Function, RegExp, XML et XMLList.

De nombreux langages de programmation font la distinction entre les valeurs primitives et leurs enveloppes. Java, par exemple, a une valeur primitive int et la classe java.lang.Integer qui l'enveloppe. Les valeurs primitives de Java ne sont pas des objets, mais leurs enveloppes le sont, ce qui rend les valeurs primitives utiles pour certaines opérations et les enveloppes pour d'autres. Dans ActionScript 3.0, les valeurs primitives et leurs enveloppes ne peuvent être distinguées, à des fins pratiques. Toutes les valeurs, même les valeurs primitives, sont des objets. Le moteur d'exécution traite ces types primitifs comme des cas spéciaux qui se comportent comme des objets n'exigeant pas le temps normal associé à la création d'objets. Cela signifie que les deux lignes de code suivantes sont équivalentes :

```
var someInt:int = 3;
var someInt:int = new int(3);
```

Tous les types de données primitifs et complexes répertoriés ci-dessus sont définis par les classes de base d'ActionScript 3.0. Les classes de base vous permettent de créer des objets à l'aide de valeurs littérales au lieu d'utiliser l'opérateur `new`. Par exemple, vous pouvez créer un tableau à l'aide d'une valeur littérale ou du constructeur de classe Array, comme suit :

```
var someArray:Array = [1, 2, 3]; // literal value
var someArray:Array = new Array(1,2,3); // Array constructor
```

Vérification des types

La vérification des types peut être effectuée lors de la compilation ou de l'exécution. Les langages typés statiquement (C++ et Java, par exemple) effectuent la vérification des types lors de la compilation. Les langages typés dynamiquement (Smalltalk et Python, par exemple) effectuent la vérification des types lors de l'exécution. En tant que langage typé dynamiquement, ActionScript 3.0 effectue la vérification des types lors de l'exécution mais également lors de la compilation, avec un mode de compilateur spécial appelé *mode strict*. En mode strict, la vérification des types a lieu à la fois lors de la compilation et de l'exécution, mais en mode standard, elle a lieu lors de l'exécution.

Les langages typés dynamiquement vous permettent de structurer votre code avec une grande souplesse, mais des erreurs de type risquent de se produire lors de l'exécution. Les langages typés statiquement signalent des erreurs de type lors de la compilation mais des informations de type sont requises à ce moment-là.

Vérification des types lors de la compilation

La vérification des types lors de la compilation est souvent favorisée dans les projets plus importants car au fur et à mesure que la taille du projet augmente, la flexibilité du type de données devient généralement moins importante que la capture d'erreurs de type aussitôt que possible. C'est pourquoi le compilateur d'ActionScript dans Flash Professional et dans Flash Builder est exécuté en mode strict, par défaut.

Adobe Flash Builder

Vous pouvez désactiver le mode strict dans Flash Builder par le biais des paramètres du compilateur d'ActionScript dans la boîte de dialogue Project Properties.

Pour que la vérification des types soit exécutée lors de la compilation, le compilateur doit connaître les informations de type de données des variables ou expressions que contient votre code. Pour déclarer explicitement un type de données pour une variable, ajoutez l'opérateur deux points (`:`) suivi du type de données comme suffixe du nom de variable. Pour associer un type de données à un paramètre, utilisez l'opérateur deux points suivi du type de données. Par exemple, le code suivant ajoute des informations de type de données au paramètre `xParam`, et déclare une variable `myParam` avec un type de données explicite :

```
function runtimeTest(xParam:String)
{
    trace(xParam);
}
var myParam:String = "hello";
runtimeTest(myParam);
```

En mode strict, le compilateur d'ActionScript signale des incompatibilités de type comme erreurs de compilateur. Par exemple, le code suivant déclare un paramètre de fonction `xParam`, de type Object, mais tente ensuite d'affecter des valeurs de type String et Number à ce paramètre. Ceci produit une erreur de compilateur en mode strict.

```
function dynamicTest(xParam:Object)
{
    if (xParam is String)
    {
        var myStr:String = xParam; // compiler error in strict mode
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam; // compiler error in strict mode
        trace("Number: " + myNum);
    }
}
```

Même en mode strict, cependant, vous pouvez choisir d'abandonner la vérification des types lors de la compilation en laissant la partie droite d'une instruction d'affectation non typée. Vous pouvez marquer une variable ou une expression comme non typée soit en omettant une annotation de type, soit à l'aide de l'annotation de type astérisque (`*`) spéciale. Par exemple, si le paramètre `xParam` de l'exemple précédent est modifié de façon à ne plus avoir d'annotation de type, la compilation s'effectue en mode strict :

```
function dynamicTest(xParam)
{
    if (xParam is String)
    {
        var myStr:String = xParam;
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam;
        trace("Number: " + myNum);
    }
}
dynamicTest(100)
dynamicTest("one hundred");
```

Vérification des types lors de l'exécution

La vérification des types lors de l'exécution s'applique dans ActionScript 3.0 quel que soit le mode, strict ou standard. Imaginez que la valeur 3 est transmise en tant qu'argument à une fonction qui attend un tableau. En mode strict, le compilateur génère une erreur car la valeur 3 n'est pas compatible avec le type de données Array. Si vous désactivez le mode strict et utilisez le mode standard, le compilateur ne signale pas l'incompatibilité de type, mais la vérification des types lors de l'exécution provoque une erreur d'exécution.

L'exemple suivant présente une fonction appelée `typeTest()` qui attend une instruction Array mais qui reçoit une valeur de 3. Ceci provoque une erreur d'exécution en mode standard car la valeur 3 n'est pas un membre du type de données déclaré (Array) du paramètre.

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0 standard mode
```

Il se peut également que vous obteniez une erreur de type lors de l'exécution alors que vous êtes en mode strict. Ceci est possible si vous utilisez le mode strict mais que vous abandonnez la vérification des types lors de la compilation à l'aide d'une variable non typée. Lorsque vous utilisez une variable non typée, vous n'éliminez pas la vérification des types mais la remettez à plus tard, au moment de l'exécution. Par exemple, si le type de données de la variable `myNum` de l'exemple précédent n'est pas déclaré, le compilateur ne peut pas détecter l'incompatibilité de type, mais le code génère une erreur d'exécution car il compare la valeur d'exécution de `myNum` (définie sur 3 en raison de l'instruction d'affectation) avec le type de `xParam` (défini sur Array).

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0
```

La vérification des types lors de l'exécution permet également d'utiliser l'héritage avec davantage de souplesse que la vérification lors de la compilation. En remettant la vérification des types au moment de l'exécution, le mode standard vous permet de référencer des propriétés d'une sous-classe même si vous effectuez un *upcast*. Un upcast a lieu lorsque vous utilisez une classe de base pour déclarer le type d'une occurrence de classe mais une sous-classe pour l'instancier. Par exemple, vous pouvez créer une classe appelée ClassBase qui peut être étendue (les classes avec l'attribut `final` ne peuvent pas être étendues) :

```
class ClassBase
{
}
```

Vous pouvez ensuite créer une sous-classe de ClassBase appelée ClassExtender, qui possède une propriété appelée `someString`, comme suit :

```
class ClassExtender extends ClassBase
{
    var someString:String;
}
```

Vous pouvez utiliser les deux classes pour créer une occurrence de classe déclarée à l'aide du type de données ClassBase mais instanciée avec le constructeur ClassExtender. Un upcast est considéré comme une opération sûre car la classe de base ne contient aucune propriété ni méthode ne se trouvant pas dans la sous-classe.

```
var myClass:ClassBase = new ClassExtender();
```

Une sous-classe, néanmoins, contient des propriétés et des méthodes que sa classe de base ne contient pas. Par exemple, la classe ClassExtender contient la propriété `someString` qui n'existe pas dans la classe ClassBase. Dans ActionScript 3.0 en mode standard, vous pouvez référencer cette propriété à l'aide de l'occurrence `myClass` sans générer d'erreur d'exécution, comme indiqué dans l'exemple suivant :

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// no error in ActionScript 3.0 standard mode
```

Opérateur is

L'opérateur `is` permet de tester si une variable ou une expression est membre d'un type de données. Dans les versions précédentes d'ActionScript, l'opérateur `instanceof` offrait cette fonctionnalité, mais dans ActionScript 3.0, l'opérateur `instanceof` ne doit pas être utilisé pour tester l'appartenance à un type de données. L'opérateur `is` doit être utilisé à la place de l'opérateur `instanceof` pour la vérification des types manuelle car l'expression `x instanceof y` vérifie simplement la chaîne de prototype de `x` pour voir si `y` existe (et dans ActionScript 3.0, la chaîne de prototype ne donne pas une image complète de la hiérarchie d'héritage).

L'opérateur `is` examine sa hiérarchie d'héritage et peut être utilisé pour vérifier non seulement si un objet est une occurrence d'une classe particulière mais également si un objet est une occurrence d'une classe qui implémente une interface particulière. L'exemple suivant crée une occurrence de la classe Sprite appelée `mySprite` et utilise l'opérateur `is` pour tester si `mySprite` est une occurrence des classes Sprite et DisplayObject, et si elle implémente l'interface IEventDispatcher :

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite); // true
trace(mySprite is DisplayObject);// true
trace(mySprite is IEventDispatcher); // true
```

L'opérateur `is` vérifie la hiérarchie d'héritage et signale correctement que `mySprite` est compatible avec les classes Sprite et DisplayObject (la classe Sprite est une sous-classe de la classe DisplayObject). L'opérateur `is` vérifie également si `mySprite` hérite d'une classe qui implémente l'interface IEventDispatcher. Etant donné que la classe Sprite hérite de la classe EventDispatcher, qui implémente l'interface IEventDispatcher, l'opérateur `is` signale correctement que `mySprite` implémente la même interface.

L'exemple suivant présente les mêmes tests que l'exemple précédent, mais avec `instanceof` au lieu de l'opérateur `is`. L'opérateur `instanceof` identifie correctement que `mySprite` est une occurrence de Sprite ou de DisplayObject, mais il renvoie `false` lorsqu'il est utilisé pour tester si `mySprite` implémente l'interface IEventDispatcher.

```
trace(mySprite instanceof Sprite); // true
trace(mySprite instanceof DisplayObject);// true
trace(mySprite instanceof IEventDispatcher); // false
```

Opérateur as

L'opérateur `as` permet également de vérifier si une expression est membre d'un type de données. Contrairement à l'opérateur `is`, cependant, l'opérateur `as` ne renvoie pas de valeur booléenne. L'opérateur `as` renvoie la valeur de l'expression au lieu de `true`, et `null` au lieu de `false`. L'exemple suivant indique les résultats obtenus si vous utilisez l'opérateur `as` au lieu de l'opérateur `is` pour vérifier simplement si une occurrence de Sprite appartient aux types de données DisplayObject, IEventDispatcher et Number.

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [object Sprite]
trace(mySprite as DisplayObject); // [object Sprite]
trace(mySprite as IEventDispatcher); // [object Sprite]
trace(mySprite as Number);          // null
```

Lorsque vous utilisez l'opérateur `as`, l'opérande à droite doit être un type de données. Vous obtenez une erreur si vous tentez d'utiliser une expression autre qu'un type de données comme opérande à droite.

Classes dynamiques

Une classe *dynamique* définit un objet qui peut être modifié lors de l'exécution en ajoutant ou en modifiant des propriétés et des méthodes. Une classe qui n'est pas dynamique (la classe String, par exemple), est une classe *scellée*. Vous ne pouvez pas ajouter de propriétés ni de méthodes à une classe scellée lors de l'exécution.

Vous créez des classes dynamiques en utilisant l'attribut `dynamic` lorsque vous déclarez une classe. Par exemple, le code suivant crée une classe dynamique appelée `Protean` :

```
dynamic class Protean
{
    private var privateGreeting:String = "hi";
    public var publicGreeting:String = "hello";
    function Protean()
    {
        trace("Protean instance created");
    }
}
```

Si vous instanciez ensuite une occurrence de la classe `Protean`, vous pouvez lui ajouter des propriétés ou des méthodes en dehors de la définition de classe. Par exemple, le code suivant crée une occurrence de la classe `Protean` et lui ajoute une propriété appelée `aString` et une autre appelée `aNumber` :

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // testing 3
```

Les propriétés que vous ajoutez à une occurrence d'une classe dynamique sont des entités d'exécution. Par conséquent, toute vérification des types est effectuée lors de l'exécution. Vous ne pouvez pas ajouter une annotation de type à une propriété que vous ajoutez de cette façon.

Vous pouvez également ajouter une méthode à l'occurrence `myProtean` en définissant une fonction et en l'associant à une propriété de l'occurrence `myProtean`. Le code suivant déplace l'instruction trace dans une méthode appelée `traceProtean()` :

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
    trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // testing 3
```

Cependant, les méthodes créées de cette manière n'ont pas accès à des méthodes ou à des propriétés privées de la classe Protean. De plus, même les références à des méthodes ou à des propriétés publiques de la classe `Protean` doivent être qualifiées avec le mot-clé `this` ou le nom de classe. L'exemple suivant présente la méthode `traceProtean()` tentant d'accéder aux variables privées et publiques de la classe `Protean`.

```
myProtean.traceProtean = function ()
{
    trace(myProtean.privateGreeting); // undefined
    trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

Descriptions des types de données

Les types de données primitifs peuvent être Boolean, int, Null, Number, String, uint et void. Les classes de base d'ActionScript définissent également les types de données complexes suivants : Object, Array, Date, Error, Function, RegExp, XML et XMLList.

Type de données Boolean

Le type de données Boolean comporte deux valeurs : `true` et `false`. Les variables du type booléen ne prennent en charge aucune autre valeur. La valeur par défaut d'une variable booléenne qui a été déclarée mais non initialisée est `false`.

Type de données int

Le type de données int est stocké en interne en tant que nombre entier 32 bits et comprend l'ensemble des entiers allant de

-2 147 483 648 (-2^{31}) à 2 147 483 647 ($2^{31} - 1$), inclus. Les versions précédentes d'ActionScript offraient uniquement le type de données Number, qui était utilisé à la fois pour les nombres entiers et les nombres en virgule flottante. Dans ActionScript 3.0, vous avez maintenant accès à des types machine de bas niveau pour les nombres entiers 32 bits signés et non signés. Si la variable ne doit pas utiliser de nombres en virgule flottante, il est plus rapide et efficace d'utiliser le type de données int plutôt que le type de données Number.

Pour les valeurs entières en dehors de la plage des valeurs int minimum et maximum, utilisez le type de données Number, qui peut gérer des valeurs positives et négatives 9 007 199 254 740 992 (valeurs entières 53 bits). La valeur par défaut pour des variables du type de données int est 0.

Type de données Null

Le type de données Null contient une seule valeur, `null`. Il s'agit de la valeur par défaut pour le type de données String et toutes les classes qui définissent des types de données complexes, y compris la classe Object. Aucun des autres types de données primitifs (Boolean, Number, int et uint) ne contient la valeur `null`. A l'exécution, la valeur `null` est convertie en la valeur par défaut appropriée si vous tentez d'affecter `null` à des variables de type Boolean, Number, int ou uint. Vous ne pouvez pas utiliser ce type de donnés comme annotation de type.

Type de données Number

Dans ActionScript 3.0, le type de données Number peut représenter des entiers, des entiers non signés et des nombres en virgule flottante. Néanmoins, pour optimiser les performances, utilisez le type de données Number uniquement pour des valeurs entières supérieures à ce que les types `int` et `uint` 32 bits peuvent stocker ou pour des nombres en virgule flottante. Pour stocker un nombre en virgule flottante, vous devez y insérer une virgule. Si vous omettez la virgule, le nombre est stocké comme nombre entier.

Le type de données Number utilise le format à deux décimales (64 bits) conformément à la norme IEEE 754. Cette norme dicte comment les nombres en virgule flottante sont stockés à l'aide des 64 bits disponibles. Un bit est utilisé pour désigner si le nombre est positif ou négatif. Onze bits sont utilisés pour l'exposant, qui est stocké comme base 2. Les 52 bits restants sont utilisés pour stocker le *significande* (également appelé *mantisse*), qui correspond au nombre élevé à la puissance indiqué par l'exposant.

En utilisant certains de ses bits pour stocker un exposant, le type de données Number peut stocker des nombres en virgule flottante beaucoup plus grands que s'il utilisait tous ses bits pour le significande. Par exemple, si le type de données Number utilisait les 64 bits pour stocker le significande, il pourrait stocker un nombre aussi grand que $2^{65} - 1$. En utilisant 11 bits pour stocker un exposant, le type de données Number peut élever son significande à une puissance de 2^{1023}.

Les valeurs maximum et minimum que le type Number peut représenter sont stockées dans des propriétés statiques de la classe Number appelées `Number.MAX_VALUE` et `Number.MIN_VALUE`.

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

Bien que cette plage de nombres soit énorme, elle est d'une grande précision. Le type de données Number utilise 52 bits pour stocker le significande. Par conséquent, les nombres nécessitant plus de 52 bits pour une représentation précise (la fraction 1/3, par exemple) ne sont que des approximations. Si votre application exige une précision absolue avec des nombres décimaux, vous devez utiliser le logiciel qui implémente l'arithmétique flottante décimale plutôt que l'arithmétique flottante binaire.

Lorsque vous stockez des valeurs entières avec le type de données Number, seuls les 52 bits du significande sont utilisés. Le type de données Number utilise ces 52 bits et un bit masqué spécial pour représenter des nombres entiers de -9 007 199 254 740 992 (-2^{53}) à 9 007 199 254 740 992 (2^{53}).

La valeur `NaN`est utilisée non seulement comme valeur par défaut pour des variables de type `Number`, mais également comme résultat de toute opération devant renvoyer un nombre mais ne le faisant pas. Par exemple, si vous tentez de calculer la racine carrée d'un nombre négatif, le résultat est `NaN`. D'autres valeurs Number spéciales comprennent l'*infini positif* et l'*infini négatif*.

Remarque : *le résultat de la division par `0` n'est `NaN` que si le diviseur est également `0`. La division par `0` produit `infinity` lorsque le dividende est positif ou `-infinity` lorsqu'il est négatif.*

Type de données String

Le type de données String représente une séquence de caractères 16 bits. Les chaînes sont stockées en interne sous forme de caractères Unicode, au format UTF-16. Les chaînes sont des valeurs inaltérables, comme dans le langage de programmation Java. Toute opération effectuée sur une valeur String renvoie une nouvelle occurrence de la chaîne. La valeur par défaut d'une variable déclarée avec le type de données String est `null`. La valeur `null` n'est pas identique à la chaîne vide (`""`). Elle signifie qu'aucune valeur n'est stockée dans la variable, alors que la chaîne vide indique que la valeur de la variable est une chaîne ne contenant pas de caractères.

Type de données uint

Le type de données uint est stocké en interne sous la forme d'un entier 32 bits non signé et est constitué de l'ensemble d'entiers compris entre 0 et 4 294 967 295 (2^{32}-1), inclus. Utilisez le type de données uint dans des situations spéciales nécessitant des entiers non négatifs. Par exemple, vous devez utiliser le type de données uint pour représenter les valeurs de couleur des pixels car le type de données int a un bit de signe interne non approprié pour la gestion des valeurs de couleur. Pour les valeurs d'entiers supérieures à la valeur uint maximale, utilisez le type de données Number qui peut gérer des valeurs d'entiers 53 bits. La valeur par défaut pour des variables du type de données uint est 0.

Type de données Void

Le type de données void contient une seule valeur, `undefined`. Dans les versions précédentes d'ActionScript, `undefined` était la valeur par défaut des occurrences de la classe Object. Dans ActionScript 3.0, la valeur par défaut des occurrences Object est `null`. Si vous tentez d'affecter la valeur `undefined` à une occurrence de la classe Object, la valeur est convertie en `null`. Vous pouvez affecter uniquement une valeur de `undefined` à des variables non typées. Les variables non typées sont des variables dépourvues de toute annotation de type, ou qui utilisent l'astérisque (`*`) pour l'annotation de type. Vous pouvez utiliser `void` uniquement comme annotation de type renvoyé.

Type de données Object

Ce type de données est défini par la classe Object. La classe Object sert de classe de base pour toutes les définitions de classe dans ActionScript. La version d'ActionScript 3.0 du type de données Object diffère de celle des versions précédentes de trois façons. Premièrement, le type de données Object n'est plus le type de données par défaut affecté aux variables sans annotation de type. Deuxièmement, le type de données Object ne comprend plus la valeur `undefined`, qui était la valeur par défaut des occurrences Object. Troisièmement, dans ActionScript 3.0, la valeur par défaut des occurrences de la classe Object est `null`.

Dans les versions précédentes d'ActionScript, une variable sans annotation de type était affectée automatiquement au type de données Object. Ceci n'est plus valable dans ActionScript 3.0, qui inclut maintenant la notion de variable parfaitement non typée. Les variables sans annotation de type sont désormais considérées comme non typées. Si vous souhaitez faire comprendre aux lecteurs de votre code que vous souhaitez laisser une variable non typée, vous pouvez utiliser l'astérisque (`*`) pour l'annotation de type, ce qui revient à omettre celle-ci. L'exemple suivant présente deux instructions équivalentes qui déclarent une variable non typée `x` :

```
var x
var x:*
```

Seules les variables non typées peuvent contenir la valeur `undefined`. Si vous tentez d'affecter la valeur `undefined` à une variable ayant un type de données, le moteur d'exécution convertit la valeur `undefined` en la valeur par défaut pour ce type de données. Pour des occurrences du type de données Object, la valeur par défaut est `null` ; autrement dit, si vous tentez d'affecter la valeur `undefined` à une occurrence d'Object, la valeur est convertie en `null`.

Conversions de type

Une conversion de type a lieu lorsqu'une valeur est transformée en une valeur d'un type de données différent. Les conversions de type peuvent être *implicites* ou *explicites*. Les conversions implicites (ou *coercition*) sont parfois effectuées lors de l'exécution. Par exemple, si la valeur 2 est affectée à une variable du type de données Boolean, elle est convertie en la valeur booléenne `true` avant son affectation. Les conversions explicites (ou *association*) ont lieu lorsque votre code demande au compilateur de traiter une variable d'un type de données comme si elle appartenait à un type de données différent. Lorsque des valeurs primitives sont impliquées, l'association convertit les valeurs d'un type de données en un autre. Pour associer un objet à un autre type de données, vous mettez le nom de l'objet entre parenthèses et le faites précéder du nom du nouveau type. Par exemple, le code suivant prend une valeur booléenne et la transforme en entier numérique :

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

Conversions implicites

Les conversions implicites ont lieu lors de l'exécution dans plusieurs contextes :

- Dans des instructions d'affectation
- Lorsque des valeurs sont transmises en tant qu'arguments de fonction
- Lorsque des valeurs sont renvoyées à partir de fonctions
- Dans les expressions utilisant certains opérateurs tels que l'opérateur d'addition (`+`)

  Pour les types définis par l'utilisateur, les conversions implicites ont lieu lorsque la valeur à convertir est une occurrence de la classe de destination ou d'une classe qui dérive de cette dernière. En cas d'échec de la conversion implicite, une erreur se produit. Par exemple, le code suivant contient une conversion implicite ayant réussi et une conversion implicite ayant échoué:

  ```
  class A {}
  class B extends A {}

  var objA:A = new A();
  var objB:B = new B();
  var arr:Array = new Array();

  objA = objB; // Conversion succeeds.
  objB = arr; // Conversion fails.
  ```

Pour les types primitifs, les conversions implicites sont gérées par appel des algorithmes de conversion internes appelés par les fonctions de conversion explicite.

Conversions explicites

Les conversions explicites, ou association, sont utiles lorsque vous compilez en mode strict et que vous ne souhaitez pas qu'une incompatibilité de types génère une erreur de compilation. Ceci peut se produire lorsque vous savez que la coercition convertira vos valeurs correctement lors de l'exécution. Par exemple, lorsque vous utilisez des données reçues d'un formulaire, vous pouvez utiliser la coercition pour convertir certaines valeurs de chaîne en valeurs numériques. Le code suivant génère une erreur de compilation même si le code s'exécute correctement en mode standard :

```
var quantityField:String = "3";
var quantity:int = quantityField; // compile time error in strict mode
```

Si vous souhaitez continuer à utiliser le mode strict, mais que vous souhaitez que la chaîne soit convertie en nombre entier, vous pouvez utiliser la conversion explicite, comme suit :

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // Explicit conversion succeeds.
```

Association à int, uint et Number

Vous pouvez associer tout type de données à l'un des trois types de nombre suivants : int, uint et Number. Si, pour une raison quelconque, la conversion du nombre est impossible, la valeur par défaut 0 est affectée pour les types de données int et uint, et la valeur par défaut `NaN` est affectée pour le type de données Number. Si vous convertissez une valeur booléenne en un nombre, `true` devient la valeur 1 et `false` devient la valeur 0.

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

Les valeurs de chaîne qui contiennent des chiffres uniquement peuvent être converties en l'un des types de nombre. Les types de nombre peuvent également convertir des chaînes ressemblant à des nombres négatifs ou des chaînes représentant une valeur hexadécimale (par exemple, `0x1A`). Le processus de conversion ignore les espaces blancs à gauche ou à droite dans la valeur de chaîne. Vous pouvez également associer des chaînes qui ressemblent à des nombres en virgule flottante à l'aide de `Number()`. Le fait d'inclure une virgule fait que `uint()` et `int()` renvoient un entier avec les caractères suivant la virgule ayant été tronqués. Par exemple, les valeurs de chaîne suivantes peuvent être associées à des nombres :

```
trace(uint("5")); // 5
trace(uint("-5")); // 4294967291. It wraps around from MAX_VALUE
trace(uint(" 27 ")); // 27
trace(uint("3.7")); // 3
trace(int("3.7")); // 3
trace(int("0x1A")); // 26
trace(Number("3.7")); // 3.7
```

Les valeurs de chaîne qui contiennent des caractères non numériques renvoient 0 lors de l'association à `int()` ou `uint()` et `NaN` lors de l'association à `Number()`. Le processus de conversion ignore les espaces blancs à gauche et à droite mais renvoie 0 ou `NaN` si une chaîne contient un espace blanc séparant deux nombres.

```
trace(uint("5a")); // 0
trace(uint("ten")); // 0
trace(uint("17 63")); // 0
```

Dans ActionScript 3.0, la fonction `Number()` ne prend plus en charge les nombres octaux, ou de base 8. Si vous fournissez une chaîne avec un zéro à gauche à la fonction `Number()` d'ActionScript 2.0, le nombre est interprété comme un nombre octal et converti en son équivalent décimal. Ceci ne s'applique pas à la fonction `Number()` dans ActionScript 3.0, qui ignore le zéro à gauche. Par exemple, le code suivant génère un résultat différent lorsqu'il est compilé à l'aide de différentes versions d'ActionScript :

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

L'association n'est pas nécessaire lorsqu'une valeur d'un type numérique est affectée à une variable d'un type numérique différent. Même en mode strict, les types numériques sont convertis de façon implicite en d'autres types numériques. Ceci signifie que dans certains cas, des valeurs inattendues peuvent être renvoyées lorsque la plage d'un type est dépassée. Les exemples suivants compilent tous en mode strict, même si certains génèrent des valeurs inattendues :

```
var myUInt:uint = -3; // Assign int/Number value to uint variable
trace(myUInt); // 4294967293

var myNum:Number = sampleUINT; // Assign int/uint value to Number variable
trace(myNum) // 4294967293

var myInt:int = uint.MAX_VALUE + 1; // Assign Number value to uint variable
trace(myInt); // 0

myInt = int.MAX_VALUE + 1; // Assign uint/Number value to int variable
trace(myInt); // -2147483648
```

Le tableau suivant récapitule les résultats de l'association au type de données Number, int, ou uint à partir d'autres types de données.

Valeur ou type de données	**Résultat de la conversion en Number, int ou uint**
Boolean	Si la valeur est `true`, 1 ; sinon, 0.
Date	Représentation interne de l'objet Date, qui est le nombre de millisecondes depuis le 1er janvier 1970, à minuit, heure universelle.
`null`	0
Object	Si l'occurrence est `null` et convertie en Number, `NaN` ; sinon, 0.
String	Un nombre si la chaîne peut être convertie en nombre ; sinon, `NaN` si elle est convertie en Number ou 0 si elle convertie en int ou uint.
`undefined`	Si converti en Number, `NaN` ; si converti en int ou uint, 0.

Association à Boolean

L'association à Boolean à partir de n'importe quel type de données numériques (uint, int et Number) donne `false` si la valeur numérique est 0, et `true` autrement. Pour le type de données Number, la valeur `NaN` donne également `false`. L'exemple suivant indique les résultats de l'association des chiffres -1, 0, et 1 :

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
    trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

Le résultat de l'exemple indique que sur les trois chiffres, seul 0 renvoie une valeur `false` :

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

L'association à Boolean à partir d'une valeur String renvoie `false` si la chaîne est `null` ou une chaîne vide (`""`). Sinon, elle renvoie `true`.

```
var str1:String; // Uninitialized string is null.
trace(Boolean(str1)); // false

var str2:String = ""; // empty string
trace(Boolean(str2)); // false

var str3:String = " "; // white space only
trace(Boolean(str3)); // true
```

L'association à Boolean à partir d'une occurrence de la classe Object renvoie `false` si l'occurrence est `null`, et `true` autrement :

```
var myObj:Object; // Uninitialized object is null.
trace(Boolean(myObj)); // false

myObj = new Object(); // instantiate
trace(Boolean(myObj)); // true
```

Les variables booléennes bénéficient d'un traitement particulier en mode strict car vous pouvez affecter des valeurs de tout type de données à une variable booléenne sans association. La coercition implicite de tous les types de données au type de données Boolean a lieu même en mode strict. En d'autres termes, contrairement à la plupart de tous les autres types de données, l'association à Boolean n'est pas nécessaire pour éviter des erreurs en mode strict. Les exemples suivants compilent tous en mode strict et se comportent comme prévu lors de l'exécution :

```
var myObj:Object = new Object(); // instantiate
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

Le tableau suivant récapitule les résultats de l'association au type de données Boolean à partir d'autres types de données :

Valeur ou type de données	Résultat de la conversion en Boolean
String	`false` si la valeur est `null` ou la chaîne vide (`""`) ; `true` autrement.
`null`	`false`
Number, int ou uint	`false` si la valeur est `NaN` ou 0 ; `true` autrement.
Object	`false` si l'occurrence est `null` ; `true` autrement.

Association à String

L'association au type de données String à partir de n'importe quel type de données numérique renvoie une représentation sous forme de chaîne du nombre. L'association au type de données String à partir d'une valeur booléenne renvoie la chaîne `"true"` si la valeur est `true`, et renvoie la chaîne `"false"` si la valeur est `false`.

L'association au type de données String à partir d'une occurrence de la classe Object renvoie la chaîne `"null"` si l'occurrence est `null`. Autrement, l'association au type String à partir de la classe Object renvoie la chaîne `"[object Object]"`.

L'association à String à partir d'une occurrence de la classe Array renvoie une chaîne comprenant une liste séparée par des virgules de tous les éléments du tableau. Par exemple, l'association suivante au type de données String renvoie une chaîne contenant les trois éléments du tableau :

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

L'association à String à partir d'une occurrence de la classe Date renvoie une représentation sous forme de chaîne de la date que l'occurrence contient. Par exemple, l'exemple suivant renvoie une représentation sous forme de chaîne de l'occurrence de la classe Date (le résultat indiqué correspond à l'heure d'été de la côte ouest des Etats-Unis) :

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // Fri Jul 1 00:00:00 GMT-0700 2005
```

Le tableau suivant récapitule les résultats de l'association au type de données String à partir d'autres types de données.

Valeur ou type de données	Résultat de la conversion en chaîne
Array	Une chaîne composée de tous les éléments du tableau
Boolean	`"true"` ou `"false"`
Date	Une représentation sous forme de chaîne de l'objet Date
`null`	`"null"`
Number, int ou uint	Une représentation sous forme de chaîne du nombre
Object	Si l'occurrence est null, `"null"` ; sinon, `"[object Object]"`.

Syntaxe

La syntaxe d'un langage définit un ensemble de règles à suivre lors de l'écriture du code exécutable.

Respect de la casse

ActionScript 3.0 est un langage qui fait la distinction entre les majuscules et les minuscules. Les identifiants qui diffèrent au niveau de la casse uniquement sont considérés comme différents. Par exemple, le code suivant crée deux variables différentes :

```
var num1:int;
var Num1:int;
```

Syntaxe à point

L'opérateur point (.) permet d'accéder aux propriétés et aux méthodes d'un objet. La syntaxe à point vous permet de vous rapporter à une méthode ou à une propriété de classe à l'aide d'un nom d'occurrence, suivi par l'opérateur point et le nom de la méthode ou de la propriété. Par exemple, considérez la définition de classe suivante :

```
class DotExample
{
    public var prop1:String;
    public function method1():void {}
}
```

La syntaxe à point vous permet d'accéder à la propriété `prop1` et à la méthode `method1()` à l'aide du nom d'occurrence créé dans le code suivant :

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = "hello";
myDotEx.method1();
```

Vous pouvez utiliser la syntaxe à point lorsque vous définissez des packages. Vous utilisez l'opérateur point pour référencer des packages imbriqués. Par exemple, la classe EventDispatcher se trouve dans un package appelé events qui est imbriqué dans le package nommé flash. Vous pouvez référencer le package events à l'aide de l'expression suivante :

```
flash.events
```

Vous pouvez également référencer la classe EventDispatcher au moyen de cette expression :

```
flash.events.EventDispatcher
```

Syntaxe à barre oblique

La syntaxe à barre oblique n'est pas prise en charge dans ActionScript 3.0. Elle était utilisée dans les versions antérieures d'ActionScript pour indiquer le chemin d'un clip ou d'une variable.

Littéraux

Un *littéral* est une valeur qui apparaît directement dans votre code. Voici quelques exemples de littéraux :

```
17
"hello"
-3
9.4
null
undefined
true
false
```

Les littéraux peuvent également être regroupés pour former des littéraux composés. Les littéraux de tableau sont placés entre crochets (`[]`) et utilisent la virgule pour séparer les éléments du tableau.

Un littéral de tableau permet d'initialiser un tableau. Les exemples suivants présentent deux tableaux initialisés par des littéraux de tableau. Vous pouvez utiliser l'instruction `new` et transmettre le littéral composé sous forme de paramètre au constructeur de classe Array, ou affecter directement les valeurs littérales lors de l'instanciation des occurrences des classes de base ActionScript suivantes : Object, Array, String, Number, int, uint, XML, XMLList et Boolean.

```
// Use new statement.
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// Assign literal directly.
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

Les littéraux permettent également d'initialiser un objet générique. Un objet générique est une occurrence de la classe Object. Les littéraux d'objet sont placés entre accolades (`{}`) et séparent les propriétés de l'objet par une virgule. Chaque propriété est déclarée avec le caractère deux-points (`:`), séparant le nom et la valeur de la propriété.

Vous pouvez créer un objet générique via l'instruction `new` et transmettre le littéral d'objet sous forme de paramètre au constructeur de classe Object ou affecter directement le littéral d'objet à l'occurrence déclarée. L'exemple suivant illustre deux méthodes différentes de créer un objet générique et de l'initialiser avec trois propriétés (`propA`, `propB` et `propC`), chacune avec des valeurs définies sur 1, 2, et 3, respectivement :

```
// Use new statement and add properties.
var myObject:Object = new Object();
myObject.propA = 1;
myObject.propB = 2;
myObject.propC = 3;

// Assign literal directly.
var myObject:Object = {propA:1, propB:2, propC:3};
```

Voir aussi

Utilisation des chaînes

Utilisation d'expressions régulières

Initialisation de variables XML

Points-virgules

Vous pouvez utiliser le point-virgule (`;`) à la fin d'une instruction. Si vous omettez ce caractère, le compilateur suppose que chaque ligne de code représente une instruction distincte. Etant donné que de nombreux programmeurs ont l'habitude d'utiliser le point-virgule pour marquer la fin d'une instruction, vous pouvez faciliter la lecture de votre code en procédant de la sorte.

L'ajout d'un point-virgule à la fin d'une instruction permet de placer plusieurs instructions sur une même ligne, mais rend la lecture de votre code plus difficile.

Parenthèses

Vous pouvez utiliser les parenthèses (`()`) de trois façons dans ActionScript 3.0. Premièrement, elles permettent de modifier l'ordre des opérations dans une expression. Les opérations regroupées à l'intérieur de parenthèses sont toujours exécutées en premier lieu. Par exemple, vous pouvez utiliser des parenthèses pour modifier l'ordre des opérations dans le code suivant :

```
trace(2 + 3 * 4); // 14
trace((2 + 3) * 4); // 20
```

Deuxièmement, vous pouvez utiliser des parenthèses avec l'opérateur virgule (`,`) pour évaluer une série d'expressions et renvoyer le résultat de l'expression finale, comme indiqué dans l'exemple suivant :

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

Troisièmement, vous pouvez utiliser des parenthèses pour transmettre un ou plusieurs paramètres à des fonctions ou à des méthodes, comme indiqué dans l'exemple suivant, qui transmet une valeur String à la fonction `trace()` :

```
trace("hello"); // hello
```

Commentaires

Le code ActionScript 3.0 prend en charge deux types de commentaires : les commentaires d'une ligne et les commentaires de plusieurs lignes. Ces mécanismes de commentaire sont semblables aux mécanismes de commentaire de C++ et Java. Le compilateur ne tient pas compte du texte marqué comme commentaire.

Les commentaires d'une ligne commencent par deux barres obliques (`//`) et continuent jusqu'à la fin de la ligne. Par exemple, le code suivant contient un commentaire d'une ligne :

```
var someNumber:Number = 3; // a single line comment
```

Les commentaires de plusieurs lignes commencent par une barre oblique et un astérisque (`/*`) et se terminent pas un astérisque et une barre oblique (`*/`).

```
/* This is multiline comment that can span
more than one line of code. */
```

Mots-clés et mots réservés

Les *mots réservés* ne peuvent pas servir d'identifiants dans votre code car leur utilisation est réservée à ActionScript. Les mots réservés comprennent les *mots lexicaux* qui sont supprimés de l'espace de noms du programme par le compilateur. Le compilateur signale une erreur si vous utilisez un mot-clé lexical comme identifiant. Le tableau suivant répertorie les mots lexicaux d'ActionScript 3.0.

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	fonction
if	implements	import	in
instanceof	interface	internal	is
native	new	null	package

private	protected	public	return
super	switch	this	throw
to	true	try	typeof
use	var	void	while
with			

Il existe un petit ensemble de mots-clés, appelés *mots-clés syntaxiques* qui peuvent servir d'identifiants, mais qui possèdent une signification spéciale dans certains contextes. Le tableau suivant répertorie les mots-clés syntaxiques d'ActionScript 3.0.

each	get	set	namespace
inclure	dynamique	final	native
override	static		

Il existe également plusieurs identifiants auxquels il est parfois fait référence sous le terme *mots réservés pour une utilisation future*. Ces identifiants ne sont pas réservés par ActionScript 3.0, même si certains risquent d'être considérés comme des mots-clés par le logiciel incorporant ActionScript 3.0. Vous pourrez peut-être utiliser un grand nombre d'entre eux dans votre code, mais Adobe vous le déconseille car il est possible qu'ils soient utilisés en tant que mots-clés dans une version ultérieure du langage.

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic
long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

Constantes

ActionScript 3.0 prend en charge l'instruction `const` que vous pouvez utiliser pour créer des constantes. Les constantes sont des propriétés dont la valeur est fixe et non modifiable. Vous pouvez affecter une valeur à une constante une seule fois, et l'affectation doit avoir lieu à proximité de la déclaration de la constante. Par exemple, si une constante est déclarée en tant que membre d'une classe, vous pouvez lui affecter une valeur uniquement dans la déclaration ou dans le constructeur de classe.

Le code suivant déclare deux constantes. La première constante, `MINIMUM`, a une valeur affectée dans l'instruction de déclaration. La seconde constante, `MAXIMUM`, a une valeur affectée dans le constructeur. Observez que cet exemple compile uniquement en mode standard car, en mode strict, il est uniquement possible d'affecter une valeur de constante lors de l'initialisation.

```
class A
{
    public const MINIMUM:int = 0;
    public const MAXIMUM:int;

    public function A()
    {
        MAXIMUM = 10;
    }
}

var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

Une erreur se produit si vous tentez d'affecter une valeur initiale à une constante de toute autre façon. Par exemple, si vous tentez de définir la valeur initiale de `MAXIMUM` en dehors de la classe, une erreur d'exécution se produit.

```
class A
{
public const MINIMUM:int = 0;
public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // run-time error
```

ActionScript 3.0 définit un large éventail de constantes à votre usage. Par convention, les constantes dans ActionScript utilisent toutes des majuscules, avec des mots séparés par le caractère de soulignement (_). Par exemple, la définition de classe MouseEvent utilise cette convention d'appellation pour ses constantes, dont chacune représente un événement lié à une action de la souris :

```
package flash.events
{
    public class MouseEvent extends Event
    {
    public static const CLICK:String = "click";
    public static const DOUBLE_CLICK:String = "doubleClick";
    public static const MOUSE_DOWN:String = "mouseDown";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...
    }
}
```

Opérateurs

Les opérateurs sont des fonctions spéciales qui utilisent un ou plusieurs opérandes et renvoient une valeur. Un *opérande* est une valeur (généralement un littéral, une variable ou une expression) utilisée par l'opérateur comme entrée. Par exemple, dans le code suivant, les opérateurs d'addition (`+`) et de multiplication (`*`) sont utilisés avec trois opérandes littéraux (2, 3 et 4) pour renvoyer une valeur. Cette valeur est ensuite utilisée par l'opérateur d'affectation (`=`) pour attribuer la valeur renvoyée, 14, à la variable `sumNumber`.

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

Les opérateurs peuvent être unaires, binaires ou ternaires. Un opérateur *unaire* utilise un opérande. Par exemple, l'opérateur d'incrémentation (`++`) est un opérateur unaire car il utilise un seul opérande. Un opérateur *binaire* utilise deux opérandes. Par exemple, l'opérateur de division (`/`) utilise deux opérandes. Un opérateur *ternaire* utilise trois opérandes. Par exemple, l'opérateur conditionnel (`?:`) utilise trois opérandes.

Certains opérateurs sont *surchargés*, ce qui signifie qu'ils se comportent différemment selon le type ou la quantité d'opérandes qui leur est transmis. L'opérateur d'addition (`+`) est un exemple d'opérateur surchargé qui se comporte différemment selon le type de données des opérandes. Si les deux opérandes sont des nombres, l'opérateur d'addition renvoie la somme des valeurs. Si les deux opérandes sont des chaînes, l'opérateur d'addition renvoie la concaténation des deux opérandes. L'exemple de code suivant indique comment l'opérateur se comporte différemment selon les opérandes :

```
trace(5 + 5); // 10
trace("5" + "5"); // 55
```

Les opérateurs peuvent également se comporter différemment selon le nombre d'opérandes fourni. L'opérateur de soustraction (`-`) est un opérateur à la fois unaire et binaire. Lorsqu'il est fourni avec un seul opérande, l'opérateur de soustraction convertit l'opérande en valeur négative et renvoie le résultat. Lorsqu'il est fourni avec deux opérandes, l'opérateur de soustraction renvoie la différence entre les opérandes. L'exemple suivant présente l'opérateur de soustraction utilisé d'abord comme opérateur unaire, puis comme opérateur binaire.

```
trace(-3); // -3
trace(7 - 2); // 5
```

Priorité et associativité des opérateurs

La priorité et l'associativité des opérateurs déterminent leur ordre de traitement. Bien qu'il soit évident, pour ceux qui connaissent bien l'arithmétique, que le compilateur traite l'opérateur de multiplication (`*`) avant l'opérateur d'addition (`+`), le compilateur a besoin d'instructions claires quant à l'ordre à appliquer aux opérateurs. L'ensemble de ces instructions est appelé *ordre de priorité des opérateurs*. ActionScript définit une priorité par défaut que l'opérateur parenthèses (`()`) permet de modifier. Par exemple, le code suivant modifie la priorité par défaut de l'exemple précédent pour forcer le compilateur à traiter l'opérateur d'addition avant l'opérateur de multiplication :

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

Il arrive que plusieurs opérateurs de même priorité apparaissent dans la même expression. Dans ce cas, le compilateur utilise les règles d'*associativité* pour identifier le premier opérateur à traiter. Tous les opérateurs binaires, sauf les opérateurs d'affectation, sont *associatifs gauche*, ce qui signifie que les opérateurs de gauche sont traités avant ceux de droite. Les opérateurs d'affectation et l'opérateur conditionnel (`?:`) sont *associatifs droit*, ce qui signifie que les opérateurs de droite sont traités avant ceux de gauche.

Prenons par exemple les opérateurs inférieur à (`<`) et supérieur à (`>`), qui ont le même ordre de priorité. Lorsque les deux opérateurs sont employés dans la même expression, celui de gauche est traité en premier puisque tous deux sont associatifs gauche. Cela signifie que les deux instructions suivantes donnent le même résultat :

```
trace(3 > 2 < 1); // false
trace((3 > 2) < 1); // false
```

L'opérateur supérieur à est traité en premier, ce qui donne une valeur `true` car l'opérande 3 est supérieur à l'opérande 2. La valeur `true` est ensuite transmise à l'opérateur inférieur à, avec l'opérande 1. Le code suivant représente cet état intermédiaire :

```
trace((true) < 1);
```

L'opérateur Inférieur à convertit la valeur `true` en la valeur numérique 1 et compare cette valeur numérique au second opérande 1 pour renvoyer la valeur `false` (la valeur 1 n'est pas inférieure à 1).

```
trace(1 < 1); // false
```

Vous pouvez modifier l'associativité gauche par défaut avec l'opérateur parenthèses. Vous pouvez demander au compilateur de traiter l'opérateur inférieur à en premier lieu en plaçant cet opérateur et son opérande entre parenthèses. L'exemple suivant utilise l'opérateur parenthèses pour produire un résultat différent en utilisant les mêmes nombres que l'exemple précédent :

```
trace(3 > (2 < 1)); // true
```

L'opérateur inférieur à est traité en premier, ce qui donne une valeur `false` car l'opérande 2 n'est pas inférieur à l'opérande 1. La valeur `false` est ensuite transmise à l'opérateur supérieur à, avec l'opérande 3. Le code suivant représente cet état intermédiaire :

```
trace(3 > (false));
```

L'opérateur supérieur à convertit la valeur `false` en la valeur numérique 0 et compare cette valeur numérique à l'autre opérande 3 pour renvoyer la valeur `true` (la valeur 3 est supérieure à 0).

```
trace(3 > 0); // true
```

Le tableau suivant répertorie les opérateurs gérés par ActionScript 3.0 par ordre de priorité décroissant. Chaque ligne du tableau contient des opérateurs ayant la même priorité. Chaque ligne d'opérateurs a une priorité supérieure à la ligne située au-dessous dans le tableau.

Groupe	**Opérateurs**
Principal	`[] {x:y} () f(x) new x.y x[y] <></> @ :: ..`
Suffixe	`x++ x--`
Unaire	`++x --x + - ~ ! delete typeof void`
Multiplication	`* / %`
Ajout	`+ -`
Décalage au niveau du bit	`<< >> >>>`
Relationnel	`< > <= >= as in instanceof is`
Egalité	`== != === !==`
AND au niveau du bit	`&`
XOR au niveau du bit	`^`
OR au niveau du bit	`|`
AND logique	`&&`
OR logique	`||`
Conditionnel	`?:`
Affectation	`= *= /= %= += -= <<= >>= >>>= &= ^= |=`
Virgule	`,`

Opérateurs principaux

Les opérateurs principaux comprennent ceux utilisés pour créer des littéraux Array et Object, regrouper des expressions, appeler des fonctions, instancier des occurrences de classe et accéder à des propriétés.

Tous les opérateurs principaux, comme répertoriés dans le tableau suivant, ont la même priorité. La mention (E4X) apparaît en regard des opérateurs qui font partie de la spécification E4X.

Opérateur	Opération effectuée
`[]`	Initialise un tableau
`{x:y}`	Initialise un objet
`()`	Regroupe des expressions
`f(x)`	Appelle une fonction
`new`	Appelle un constructeur
`x.y x[y]`	Accède à une propriété
`<></>`	Initialise un objet XMLList (E4X)
`@`	Accède à un attribut (E4X)
`::`	Qualifie un nom (E4X)
`..`	Accède à un élément XML descendant (E4X)

Opérateurs de suffixe

Les opérateurs de suffixe prennent un opérateur et incrémentent ou décrémentent sa valeur. Bien que ces opérateurs soient des opérateurs unaires, ils sont classés à part du fait de leur priorité supérieure et de leur comportement particulier. Lorsque vous utilisez un opérateur de suffixe dans une expression plus grande, la valeur de l'expression est renvoyée avant le traitement de cet opérateur. Par exemple, le code suivant montre comment la valeur de l'expression `xNum++` est renvoyée avant l'incrémentation de la valeur :

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum); // 1
```

Tous les opérateurs de suffixe, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`++`	Incrémentation (suffixe)
`--`	Décrémentation (suffixe)

Opérateurs unaires

Les opérateurs unaires prennent un opérande. Les opérateurs d'incrémentation (`++`) et de décrémentation (`--`) de ce groupe sont des *opérateurs* de *préfixe*, c'est-à-dire qu'ils apparaissent avant l'opérande dans une expression. Les opérateurs de préfixe diffèrent de leur équivalent suffixe car l'opération d'incrémentation ou de décrémentation est effectuée avant le renvoi de la valeur de l'expression globale. Par exemple, le code suivant montre comment la valeur de l'expression `++xNum` est renvoyée après l'incrémentation de la valeur :

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum); // 1
```

Tous les opérateurs unaires, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`++`	Incrémentation (préfixe)
`--`	Décrémentation (préfixe)
`+`	Unaire +
`-`	Unaire - (négation)
`!`	NOT logique
`~`	NOT au niveau du bit
`delete`	Supprime une propriété
`typeof`	Renvoie les informations de type
`void`	Renvoie une valeur non définie

Opérateurs de multiplication

Les opérateurs de multiplication prennent deux opérandes et effectuent des multiplications, des divisions ou des calculs de modulo.

Tous les opérateurs de multiplication, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`*`	Multiplication
`/`	Division
`%`	Modulo

Opérateurs d'ajout

Les opérateurs d'ajout prennent deux opérandes et effectuent des calculs d'addition ou de soustraction. Tous les opérateurs d'ajout, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`+`	Addition
`-`	Soustraction

Opérateurs de décalage au niveau du bit

Ces opérateurs prennent deux opérandes et décalent les bits du premier selon la valeur spécifiée dans le second. Tous les opérateurs de décalage au niveau du bit, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`<<`	Décalage gauche au niveau du bit
`>>`	Décalage droit au niveau du bit
`>>>`	Décalage droit non signé au niveau du bit

Opérateurs relationnels

Les opérateurs relationnels prennent deux opérandes, comparent leurs valeurs et renvoient une valeur booléenne. Tous les opérateurs relationnels, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`<`	Inférieur à
`>`	Supérieur à
`<=`	Inférieur ou égal à
`>=`	Supérieur ou égal à
`as`	Vérifie le type de données
`in`	Vérifie les propriétés des objets
`instanceof`	Vérifie la chaîne de prototype
`is`	Vérifie le type de données

Opérateurs d'égalité

Les opérateurs d'égalité prennent deux opérandes, comparent leurs valeurs et renvoient une valeur booléenne. Tous les opérateurs d'égalité, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`==`	Egalité
`!=`	Inégalité
`===`	Egalité stricte
`!==`	Inégalité stricte

Opérateurs logiques au niveau du bit

Ces opérateurs prennent deux opérandes et effectuent des opérations logiques au niveau des bits. La priorité de ces opérateurs diffère et ils sont présentés dans le tableau suivant par ordre décroissant de priorité :

Opérateur	Opération effectuée
`&`	AND au niveau du bit
`^`	XOR au niveau du bit
`|`	OR au niveau du bit

Opérateurs logiques

Les opérateurs logiques prennent deux opérandes et renvoient une valeur booléenne. La priorité de ces opérateurs diffère et ils sont présentés dans le tableau suivant par ordre décroissant de priorité :

Opérateur	Opération effectuée
`&&`	AND logique
`||`	OR logique

Opérateur conditionnel

L'opérateur conditionnel est un opérateur ternaire, c'est-à-dire qu'il prend trois opérandes. Il correspond à une méthode abrégée de l'application de l'instruction conditionnelle `if..else`.

Opérateur	Opération effectuée
`?:`	Conditionnel

Opérateurs d'affectation

Les opérateurs d'affectation prennent deux opérandes et affectent une valeur à l'un d'eux en fonction de la valeur de l'autre. Tous les opérateurs d'affectation, comme répertoriés dans le tableau suivant, ont la même priorité :

Opérateur	Opération effectuée
`=`	Affectation
`*=`	Affectation de multiplication
`/=`	Affectation de division
`%=`	Affectation modulo
`+=`	Affectation d'addition
`-=`	Affectation de soustraction
`<<=`	Affectation de décalage gauche au niveau du bit
`>>=`	Affectation de décalage droit au niveau du bit
`>>>=`	Affectation de décalage droit au niveau du bit non signé
`&=`	Affectation AND au niveau du bit
`^=`	Affectation XOR au niveau du bit
`|=`	Affectation OR au niveau du bit

Instructions conditionnelles

ActionScript 3.0 fournit trois instructions conditionnelles de base que vous pouvez utiliser pour contrôler le flux du programme.

if..else

L'instruction conditionnelle `if..else` permet de tester une condition, puis d'exécuter un bloc de code lorsque cette condition est positive et d'en exécuter un autre dans le cas contraire. Par exemple, le code suivant vérifie si la valeur de `x` est supérieure à 20 et génère une fonction `trace()` dans l'affirmative ou une autre fonction `trace()` dans le cas contraire :

```
if (x > 20)
{
    trace("x is > 20");
}
else
{
    trace("x is <= 20");
}
```

Si vous ne souhaitez pas exécuter un autre bloc de code, vous pouvez utiliser l'instruction `if` sans l'instruction `else`.

if..else if

L'instruction conditionnelle `if..else if` permet de tester plusieurs conditions. Par exemple, le code suivant teste non seulement si la valeur de `x` est supérieure à 20, mais également si la valeur de `x` est négative :

```
if (x > 20)
{
    trace("x is > 20");
}
else if (x < 0)
{
    trace("x is negative");
}
```

Si une instruction `if` ou `else` est suivie d'une seule instruction, il est inutile de la placer entre accolades. Par exemple, le code suivant ne contient pas d'accolades :

```
if (x > 0)
    trace("x is positive");
else if (x < 0)
    trace("x is negative");
else
    trace("x is 0");
```

Néanmoins, Adobe conseille de toujours utiliser des accolades car un comportement inattendu peut se produire si des instructions sont ajoutées ultérieurement à une instruction conditionnelle sans accolades. Par exemple, dans le code suivant, la valeur de `positiveNums` augmente de 1 que la condition renvoie `true` ou non :

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
    trace("x is positive");
    positiveNums++;

trace(positiveNums); // 1
```

switch

L'instruction `switch` est utile si vous avez plusieurs chemins d'exécution sur la même expression de condition. Elle s'apparente à une longue série d'instructions `if..else if`, mais est plus lisible. Au lieu de tester une condition pour une valeur booléenne, l'instruction `switch` évalue une expression et utilise le résultat pour déterminer le bloc de code à exécuter. Les blocs de code commencent par une instruction `case` et se terminent par une instruction `break`. Par exemple, l'instruction `switch` suivante imprime le jour de la semaine en fonction du numéro du jour renvoyé par la méthode `Date.getDay()` :

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
    case 0:
        trace("Sunday");
        break;
    case 1:
        trace("Monday");
        break;
    case 2:
        trace("Tuesday");
        break;
    case 3:
        trace("Wednesday");
        break;
    case 4:
        trace("Thursday");
        break;
    case 5:
        trace("Friday");
        break;
    case 6:
        trace("Saturday");
        break;
    default:
        trace("Out of range");
        break;
}
```

Boucles

Les instructions en boucle vous permettent d'exécuter un bloc de code spécifique de façon répétée à l'aide d'une série de valeurs ou de variables. Adobe recommande de toujours placer le bloc de code entre accolades ({}). Vous pouvez les omettre si le bloc de code contient une seule instruction mais, comme pour les instructions conditionnelles, cette pratique est déconseillée : il est en effet possible que des instructions ajoutées à une date ultérieure soient exclues du bloc de texte par inadvertance. Si vous ajoutez ultérieurement une instruction à inclure dans le bloc de code mais que vous oubliez d'ajouter les accolades nécessaires, l'instruction n'est pas exécutée dans la boucle.

for

La boucle `for` vous permet de faire une itération sur une variable pour une plage de valeurs spécifique. Vous devez indiquer trois expressions dans une instruction `for` : une variable définie sur une valeur initiale, une instruction conditionnelle qui détermine le moment où la boucle prend fin et une expression qui modifie la valeur de la variable avec chaque boucle. Par exemple, le code suivant boucle à cinq reprises. La valeur de la variable `i` commence à 0 et prend fin à 4, et le résultat est constitué par les nombres compris entre 0 et 4, chacun sur sa propre ligne.

```
var i:int;
for (i = 0; i < 5; i++)
{
    trace(i);
}
```

for..in

La boucle `for..in` permet de faire une itération sur les propriétés d'un objet ou les éléments d'un tableau. Par exemple, utilisez la boucle `for..in` pour faire une itération sur les propriétés d'un objet générique (les propriétés d'un objet n'étant pas conservées dans un ordre particulier, elles peuvent apparaître dans un ordre imprévisible) :

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
    trace(i + ": " + myObj[i]);
}
// output:
// x: 20
// y: 30
```

Vous pouvez également faire une itération sur les éléments d'un tableau :

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
{
    trace(myArray[i]);
}
// output:
// one
// two
// three
```

En revanche, il est impossible d'itérer sur les propriétés d'un objet lorsqu'il s'agit d'une occurrence de classe scellée (classe intégrée ou définie par l'utilisateur). Pour ce faire, la classe doit être dynamique et, même dans ce cas, vous ne pouvez faire d'itération que sur les propriétés ajoutées dynamiquement.

for each..in

La boucle `for each..in` vous permet de faire une itération sur les éléments d'une collection (balises dans un objet XML ou XMLList, valeurs des propriétés d'un objet ou éléments d'un tableau). Ainsi, comme l'illustre l'extrait suivant, vous pouvez utiliser une boucle `for each..in` pour itérer sur les propriétés d'un objet générique mais, contrairement à la boucle `for..in`, la variable d'itération d'une boucle `for each..in` contient la valeur de la propriété plutôt que le nom de celle-ci :

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
    trace(num);
}
// output:
// 20
// 30
```

Vous pouvez faire une itération sur un objet XML ou XMLList, comme l'indique l'exemple suivant :

```
var myXML:XML = <users>
    <fname>Jane</fname>
    <fname>Susan</fname>
    <fname>John</fname>
</users>;

for each (var item in myXML.fname)
{
    trace(item);
}
/* output
Jane
Susan
John
*/
```

Vous pouvez également faire une itération sur les éléments d'un tableau, comme l'indique cet exemple :

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
    trace(item);
}
// output:
// one
// two
// three
```

Vous ne pouvez pas faire une itération sur les propriétés d'un objet si ce dernier est une occurrence d'une classe scellée. Même pour les occurrences de classes dynamiques, vous ne pouvez pas faire une itération sur des propriétés fixes qui sont des propriétés définies comme faisant partie d'une définition de classe.

while

La boucle `while` est semblable à une instruction `if` qui se répète tant que la condition est `true`. Par exemple, le code suivant produit le même résultat que l'exemple de boucle `for` :

```
var i:int = 0;
while (i < 5)
{
    trace(i);
    i++;
}
```

L'un des inconvénients que présente la boucle `while` par rapport à la boucle `for` est que les risques de boucles sans fin sont plus importants avec les boucles `while`. Par exemple, le code qui utilise la boucle `for` ne passe pas la compilation si vous omettez l'expression qui incrémente la variable du compteur, alors que le code qui utilise la boucle `while` est compilé. Et sans l'expression qui incrémente `i`, la boucle se poursuit sans fin.

do..while

La boucle `do..while` est une boucle `while` qui garantit que le bloc de code est exécuté au moins une fois, car la condition est vérifiée une fois que le bloc de code est exécuté. Le code suivant est un exemple simple d'une boucle `do..while` qui renvoie une sortie même si la condition n'est pas remplie.

```
var i:int = 5;
do
{
    trace(i);
    i++;
} while (i < 5);
// output: 5
```

Fonctions

Les *fonctions* sont des blocs de code qui effectuent des tâches spécifiques et qui peuvent être réutilisés dans votre programme. Il existe deux types de fonctions dans ActionScript 3.0 : les *méthodes* et les *fonctions closure*. Une fonction est appelée méthode ou fonction closure selon le contexte dans lequel elle est définie. Une fonction est appelée méthode si vous la définissez comme partie d'une définition de classe ou l'associez à l'occurrence d'un objet. Une fonction est appelée fonction closure si elle est définie de toute autre façon.

Les fonctions ont toujours été très importantes dans ActionScript. Dans ActionScript 1.0, par exemple, le mot-clé `class` n'existait pas. Par conséquent, les classes étaient définies par des fonctions constructeurs. Même si le mot-clé `class` a depuis été ajouté au langage, une solide compréhension des fonctions reste importante si vous souhaitez bénéficier de tous les avantages du langage. Ceci peut être un défi pour les programmeurs qui s'attendent à ce que les fonctions ActionScript se comportent de façon identique à celles des langages tels que C++ ou Java. Même si l'appel et la définition des fonctions de base ne devraient pas constituer un défi pour les programmeurs expérimentés, certaines des fonctions ActionScript les plus avancées nécessitent une explication.

Concepts des fonctions de base

Appel de fonctions

Vous appelez une fonction en utilisant son identifiant suivi de l'opérateur parenthèses (`()`). Vous placez les paramètres de fonction que vous souhaitez envoyer à la fonction entre parenthèses à l'aide de l'opérateur parenthèses. Par exemple, la fonction `trace()` est une fonction de haut niveau dans ActionScript 3.0 :

```
trace("Use trace to help debug your script");
```

Si vous appelez une fonction sans paramètres, vous devez utiliser une paire de parenthèses vide. Par exemple, vous pouvez utiliser la méthode `Math.random()`, qui ne prend aucun paramètre, pour générer un nombre aléatoire :

```
var randomNum:Number = Math.random();
```

Définition de vos fonctions

Dans ActionScript 3.0, deux techniques vous permettent de définir une fonction : vous pouvez utiliser une instruction de fonction ou une expression de fonction. La technique que vous choisissez dépend du style de programmation que vous préférez, plus statique ou dynamique. Définissez vos fonctions à l'aide d'instructions de fonction si vous préférez une programmation en mode strict, ou statique. Définissez vos fonctions à l'aide d'expressions de fonction si vous en avez vraiment besoin. Les expressions de fonction sont utilisées plus souvent dans la programmation en mode standard, ou dynamique.

Instructions de fonction

Les instructions de fonction représentent la technique privilégiée pour définir des fonctions en mode strict. Une instruction de fonction commence par le mot-clé `function`, suivi :

- du nom de la fonction ;
- des paramètres, dans une liste séparée par des virgules, placée entre parenthèses ;
- du corps de la fonction, c'est-à-dire le code ActionScript à exécuter lors de l'appel de la fonction, placé entre accolades.

Par exemple, le code suivant crée une fonction qui définit un paramètre puis appelle la fonction à l'aide de la chaîne « `hello` » comme valeur de paramètre :

```
function traceParameter(aParam:String)
{
    trace(aParam);
}

traceParameter("hello"); // hello
```

Expressions de fonction

La deuxième façon de déclarer une fonction est d'utiliser une instruction d'affectation avec une expression de fonction, parfois appelée littéral de fonction ou fonction anonyme. Il s'agit d'une méthode plus détaillée largement utilisée dans les versions précédentes d'ActionScript.

Une instruction d'affectation associée à une expression de fonction commence par le mot-clé `var`, suivi :

- du nom de la fonction ;
- de l'opérateur deux points (`:`) ;
- de la classe `Function` pour indiquer le type de données ;
- de l'opérateur d'affectation (`=`) ;
- du mot-clé `function` ;
- des paramètres, dans une liste séparée par des virgules, placée entre parenthèses ;
- du corps de la fonction, c'est-à-dire le code ActionScript à exécuter lors de l'appel de la fonction, placé entre accolades.

  Par exemple, le code suivant déclare la fonction `traceParameter` à l'aide d'une expression de fonction :

  ```
  var traceParameter:Function = function (aParam:String)
  {
      trace(aParam);
  };
  traceParameter("hello"); // hello
  ```

  Vous remarquerez que vous ne spécifiez pas de nom de fonction, comme dans une instruction de fonction. Une autre différence importante entre les expressions de fonction et les instructions de fonction est qu'une expression de fonction est plus une expression qu'une instruction. Ceci signifie qu'une expression de fonction ne peut pas être utilisée seule, contrairement à une instruction de fonction. Une expression de fonction peut être utilisée uniquement en tant que partie d'une instruction, généralement une instruction d'affectation. L'exemple suivant représente une expression de fonction affectée à un élément de tableau :

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
    trace(aParam);
};
traceArray[0]("hello");
```

Choix d'instructions ou d'expressions

En règle générale, utilisez une instruction de fonction, à moins que des circonstances spécifiques n'exigent l'utilisation d'une expression. Les instructions de fonction sont moins détaillées et renforcent plus la cohérence entre le mode strict et le mode standard que les expressions de fonction.

Les instructions de fonction sont plus lisibles que les instructions d'affectation qui contiennent des expressions de fonction. Les instructions de fonction rendent votre code plus concis ; elles prêtent moins à confusion que les expressions de fonction, qui exigent l'utilisation des mots-clés `var` et `function`.

Les instructions de fonction renforcent la cohérence entre les deux modes de compilateur car vous pouvez utiliser la syntaxe à point en mode strict et en mode standard pour appeler une méthode déclarée à l'aide d'une instruction de fonction. Ceci ne s'applique pas nécessairement aux méthodes déclarées avec une expression de fonction. Par exemple, le code suivant définit la classe Example à l'aide de deux méthodes : `methodExpression()`, qui est déclarée par le biais d'une expression de fonction, et `methodStatement()`, qui est déclarée par le biais d'une instruction de fonction. En mode strict, vous ne pouvez pas utiliser la syntaxe à point pour appeler la méthode `methodExpression()`.

```
class Example
{
var methodExpression = function() {}
function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

Les expressions de fonction sont plus adaptées à la programmation ciblée sur un comportement d'exécution ou dynamique. Si vous préférez utiliser le mode strict mais que vous souhaitez également appeler une méthode déclarée avec une expression de fonction, vous pouvez utiliser l'une des deux techniques. Premièrement, vous pouvez appeler la méthode à l'aide de l'opérateur crochets (`[]`) au lieu de l'opérateur point (.). L'appel de méthode suivant a lieu à la fois en mode strict et en mode standard :

```
myExample["methodLiteral"]();
```

Deuxièmement, vous pouvez déclarer la classe entière comme classe dynamique. Même si ceci vous permet d'appeler la méthode à l'aide de l'opérateur point, l'inconvénient est que vous sacrifiez une fonctionnalité de mode strict pour toutes les occurrences de cette classe. Par exemple, le compilateur ne génère pas d'erreur si vous tentez d'accéder à une propriété non définie sur une occurrence d'une classe dynamique.

Les expressions de fonction peuvent être utiles dans certains cas. Elles sont couramment utilisées pour des fonctions qui sont utilisées une seule fois. Elles peuvent être utilisées également pour associer une fonction à une propriété de prototype. Pour plus d'informations, voir Objet prototype.

Il existe deux légères différences entre les instructions de fonction et les expressions de fonction dont il faut tenir compte lorsque vous choisissez la technique à utiliser. La première réside dans le fait que les expressions de fonction n'existent pas indépendamment en tant qu'objets en ce qui concerne la gestion de la mémoire et le nettoyage. En d'autres termes, lorsque vous affectez une expression de fonction à un autre objet (un élément de tableau ou une propriété d'objet, par exemple) vous créez l'unique référence à cette expression de fonction dans votre code. Si le tableau ou l'objet auquel est associée l'expression de fonction n'est plus disponible, vous n'avez plus accès à l'expression de fonction. Si le tableau ou l'objet est supprimé, la mémoire que l'expression de fonction utilise peut être nettoyée, ce qui signifie qu'elle peut être réutilisée à d'autres fins.

L'exemple suivant indique que pour une expression de fonction, la fonction n'est plus disponible une fois que la propriété à laquelle l'expression est affectée est supprimée. La classe Test est dynamique, ce qui signifie que vous pouvez ajouter une propriété appelée `functionExp` qui contient une expression de fonction. La fonction `functionExp()` peut être appelée avec l'opérateur point, mais une fois que la propriété `functionExp` est supprimée, la fonction n'est plus accessible.

```
dynamic class Test {}
var myTest:Test = new Test();

// function expression
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp(); // Function expression
delete myTest.functionExp;
myTest.functionExp(); // error
```

Si, en revanche, la fonction est d'abord définie avec une instruction de fonction, elle existe comme son propre objet et continue à exister, même une fois que vous avez supprimé la propriété à laquelle elle est associée. L'opérateur `delete` fonctionne uniquement sur les propriétés d'objets, donc même un appel à supprimer la fonction `stateFunc()` ne fonctionne pas.

```
dynamic class Test {}
var myTest:Test = new Test();

// function statement
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // Function statement
delete myTest.statement;
delete stateFunc; // no effect
stateFunc();// Function statement
myTest.statement(); // error
```

La deuxième différence entre les instructions de fonction et les expressions de fonction réside dans le fait que les instructions de fonction existent dans le cadre dans lequel elles sont définies, notamment les instructions qui apparaissent avant l'instruction de fonction. Les expressions de fonction, en revanche, sont définies uniquement pour les instructions ultérieures. Par exemple, le code suivant appelle la fonction `scopeTest()` avant qu'elle soit définie :

```
statementTest(); // statementTest

function statementTest():void
{
    trace("statementTest");
}
```

Les expressions de fonction ne sont pas disponibles avant d'être définies. Par conséquent, le code suivant provoque une erreur d'exécution :

```
expressionTest(); // run-time error

var expressionTest:Function = function ()
{
    trace("expressionTest");
}
```

Renvoi de valeurs des fonctions

Pour renvoyer une valeur de votre fonction, utilisez l'instruction `return` suivie de l'expression ou de la valeur littérale que vous souhaitez renvoyer. Par exemple, le code suivant renvoie une expression représentant le paramètre :

```
function doubleNum(baseNum:int):int
{
    return (baseNum * 2);
}
```

Vous remarquerez que l'instruction `return` termine la fonction, de sorte que les instructions suivant une instruction `return` ne sont pas exécutées, comme suit :

```
function doubleNum(baseNum:int):int {
    return (baseNum * 2);
    trace("after return"); // This trace statement will not be executed.
}
```

En mode strict, vous devez renvoyer une valeur du type approprié si vous choisissez de spécifier un type de renvoi. Par exemple, le code suivant génère une erreur en mode strict car il ne renvoie pas de valeur valide :

```
function doubleNum(baseNum:int):int
{
    trace("after return");
}
```

Fonctions imbriquées

Vous pouvez imbriquer des fonctions, ce qui signifie que vous pouvez déclarer des fonctions au sein d'autres fonctions. Une fonction imbriquée est disponible uniquement dans sa fonction parent, à moins qu'une référence à la fonction ne soit transmise à un code externe. Par exemple, le code suivant déclare deux fonctions imbriquées à l'intérieur de la fonction `getNameAndVersion()` :

```
function getNameAndVersion():String
{
    function getVersion():String
    {
        return "10";
    }
    function getProductName():String
    {
        return "Flash Player";
    }
    return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 10
```

Lorsque des fonctions imbriquées sont transmises à un code externe, elles le sont en tant que fonctions closure, ce qui signifie que la fonction conserve les définitions se trouvant dans le domaine au moment de la définition de la fonction. Pour plus d'informations, voir Domaine d'une fonction.

Paramètres de fonction

ActionScript 3.0 permet d'exploiter des paramètres de fonction qui peuvent sembler nouveaux pour les programmeurs qui découvrent le langage. Bien que la plupart des programmeurs connaissent le principe de transfert de paramètres par valeur ou référence, l'objet `arguments` et le paramètre ... (rest) seront peut-être des nouveautés.

Transfert d'arguments par valeur ou par référence

Dans de nombreux langages de programmation, il est important de comprendre la différence entre le transfert d'arguments par valeur ou par référence car elle peut affecter la façon dont le code est conçu.

Transférer par valeur signifie que la valeur de l'argument est copiée dans une variable locale pour être utilisée dans la fonction. Transférer par référence signifie que seule une référence à l'argument est transmise, au lieu de la valeur réelle. Aucune copie de l'argument réel n'est effectuée. A la place, une référence à la variable transférée en tant qu'argument est créée et affectée à une variable locale pour être utilisée dans la fonction. En tant que référence à une variable en dehors de la fonction, la variable locale vous permet de modifier la valeur de la variable d'origine.

Dans ActionScript 3.0, tous les arguments sont transférés par référence car toutes les valeurs sont stockées en tant qu'objets. Néanmoins, les objets qui appartiennent aux types de données primitifs (Boolean, Number, int, uint et String) possèdent des opérateurs spéciaux qui font qu'ils se comportent comme s'ils étaient transférés par valeur. Par exemple, le code suivant crée une fonction appelée `passPrimitives()` qui définit deux paramètres appelés `xParam` et `yParam` de type int. Ces paramètres sont identiques aux variables locales déclarées dans le corps de la fonction `passPrimitives()`. Lorsque la fonction est appelée avec les arguments `xValue` et `yValue`, les paramètres `xParam` et `yParam` sont initialisés avec des références aux objets int représentés par `xValue` et `yValue`. Les arguments se comportent comme s'ils étaient transférés par valeur car ils sont primitifs. Bien que `xParam` et `yParam` contiennent initialement des références aux objets `xValue` et `yValue` uniquement, toute modification apportée aux variables dans le corps de fonction génère de nouvelles copies des valeurs dans la mémoire.

```
function passPrimitives(xParam:int, yParam:int):void
{
    xParam++;
    yParam++;
    trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);// 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);// 10 15
```

Dans la fonction `passPrimitives()`, les valeurs de `xParam` et `yParam` sont incrémentées, mais ceci n'affecte pas les valeurs de `xValue` et `yValue`, comme indiqué dans la dernière instruction `trace`. Ceci s'applique même si les paramètres portent les mêmes noms que les variables, `xValue` et `yValue`, car les `xValue` et `yValue` se trouvant dans la fonction pointeraient vers de nouveaux emplacements dans la mémoire qui existent indépendamment des variables du même nom en dehors de la fonction.

Tous les autres objets (c'est-à-dire les objets qui n'appartiennent pas aux types de données primitifs) sont toujours transférés par référence, ce qui vous permet de modifier la valeur de la variable d'origine. Par exemple, le code suivant crée un objet appelé `objVar` avec deux propriétés, `x` et `y`. L'objet est transféré en tant qu'argument à la fonction `passByRef()`. Etant donné que l'objet n'est pas un type primitif, non seulement il est transféré par référence mais il reste également une référence. Les changements apportés aux paramètres dans la fonction affectent donc les propriétés d'objet en dehors de la fonction.

```
function passByRef(objParam:Object):void
{
    objParam.x++;
    objParam.y++;
    trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar); // 11 16
trace(objVar.x, objVar.y); // 11 16
```

Le paramètre `objParam` référence le même objet que la variable globale `objVar`. Comme vous pouvez le constater dans les instructions `trace` de l'exemple, les modifications apportées aux propriétés `x` et `y` de l'objet `objParam` sont visibles dans l'objet `objVar`.

Valeurs de paramètre par défaut

Dans ActionScript 3.0, vous pouvez déclarer des *valeurs de paramètre par défaut* pour une fonction. Si un appel à une fonction avec des valeurs de paramètre par défaut omet un paramètre avec des valeurs par défaut, la valeur spécifiée dans la définition de fonction pour ce paramètre est utilisée. Tous les paramètres avec des valeurs par défaut doivent être placés à la fin de la liste des paramètres. Les valeurs affectées comme valeurs par défaut doivent être des constantes de compilation. L'existence d'une valeur par défaut pour un paramètre le rend *facultatif*. Un paramètre sans valeur par défaut est considéré comme un *paramètre obligatoire*.

Par exemple, le code suivant crée une fonction avec trois paramètres, dont deux possèdent des valeurs par défaut. Lorsque la fonction est appelée avec un seul paramètre, les valeurs par défaut des paramètres sont utilisées.

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
    trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

Objet arguments

Lorsque les paramètres sont transférés à une fonction, vous pouvez utiliser l'objet `arguments` pour accéder aux informations concernant les paramètres transférés à votre fonction. Voici certains aspects importants de l'objet `arguments` :

- L'objet `arguments` est un tableau qui comprend tous les paramètres transférés à la fonction.
- La propriété `arguments.length` indique le nombre de paramètres transmis à la fonction.
- La propriété `arguments.callee` fournit une référence à la fonction elle-même, ce qui est utile pour les appels récursifs à des expressions de fonction.

  Remarque : *l'objet `arguments` n'est pas disponible si un paramètre est appelé `arguments` ou si vous utilisez le paramètre ... (rest).*

  Si l'objet `arguments` est référencé dans le corps d'une fonction, ActionScript 3.0 permet aux appels de fonction d'inclure plus de paramètres que ceux définis dans la définition de fonction. Mais il génère une erreur de compilateur en mode strict si le nombre de paramètres ne correspond pas au nombre de paramètres obligatoires (et, éventuellement, au nombre de paramètres facultatifs). Vous pouvez utiliser l'aspect de tableau de l'objet `arguments` pour accéder aux paramètres transférés à la fonction, que ces paramètres soient définis ou non dans la définition de fonction. L'exemple suivant, qui est uniquement compilé en mode standard, utilise le tableau `arguments` et la propriété `arguments.length` pour suivre tous les paramètres transférés à la fonction `traceArgArray()` :

```
function traceArgArray(x:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

La propriété `arguments.callee` est souvent utilisée dans des fonctions anonymes pour créer une récursivité. Vous pouvez l'utiliser pour ajouter de la flexibilité à votre code. Si le nom de la fonction récursive change pendant votre cycle de développement, il est inutile de modifier l'appel récursif dans le corps de votre fonction si vous utilisez `arguments.callee` au lieu du nom de fonction. La propriété `arguments.callee` est utilisée dans l'expression de fonction suivante pour activer la récursivité :

```
var factorial:Function = function (x:uint)
{
    if(x == 0)
    {
        return 1;
    }
    else
    {
        return (x * arguments.callee(x - 1));
    }
}

trace(factorial(5)); // 120
```

Si vous utilisez le paramètre ... (rest) dans la déclaration de fonction, l'objet `arguments` n'est pas disponible. Vous devez accéder aux paramètres à l'aide des noms de paramètre que vous avez déclarés.

N'utilisez pas la chaîne `"arguments"` comme nom de paramètre car elle masque l'objet `arguments`. Par exemple, si la fonction `traceArgArray()` est réécrite de façon à ce qu'un paramètre `arguments` soit ajouté, les références à `arguments` dans le corps de la fonction se réfèrent au paramètre plutôt qu'à l'objet `arguments`. Le code suivant donne le résultat :

```
function traceArgArray(x:int, arguments:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// no output
```

L'objet `arguments` des versions précédentes d'ActionScript contenait également une propriété appelée `caller`, qui est une référence à la fonction qui appelait la fonction actuelle. La propriété `caller` n'existe pas dans ActionScript 3.0, mais si vous avez besoin d'une référence à la fonction d'appel, vous pouvez modifier celle-ci de façon à ce qu'elle transfère un paramètre supplémentaire qui en soit une référence.

Paramètre ... (rest)

ActionScript 3.0 présente une nouvelle déclaration de paramètre appelée le paramètre .. (rest). Ce paramètre vous permet de spécifier un paramètre de tableau qui accepte n'importe quel nombre d'arguments séparés par des virgules. Veillez à ne pas inclure un mot réservé dans le nom du paramètre. Cette déclaration de paramètre doit être le dernier paramètre spécifié. Ce paramètre rend l'objet `arguments` non disponible. Bien que le paramètre ... (rest) offre la même fonctionnalité que le tableau `arguments` et la propriété `arguments.length`, il ne fournit pas la même fonctionnalité que la propriété `arguments.callee`. Vérifiez que vous n'avez pas besoin d'utiliser `arguments.callee` avant d'utiliser le paramètre ... (rest).

L'exemple suivant réécrit la fonction `traceArgArray()` à l'aide du paramètre ... (rest) plutôt que de l'objet `arguments` :

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

Le paramètre ... (rest) peut également être utilisé avec d'autres paramètres, sous réserve qu'il soit le dernier de la liste. L'exemple suivant modifie la fonction `traceArgArray()` de façon à ce que son premier paramètre, `x`, soit de type int, et que le second utilise le paramètre ... (rest). Le résultat ignore la première valeur car le premier paramètre ne fait plus partie du tableau créé par le paramètre ... (rest).

```
function traceArgArray(x: int, ... args)
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

Fonctions comme objets

Dans ActionScript 3.0, les fonctions sont des objets. Lorsque vous créez une fonction, vous créez un objet qui peut non seulement être transmis en tant que paramètre à une autre fonction, mais auquel sont également associées des propriétés et des méthodes.

Les fonctions transférées en tant qu'arguments à une autre fonction sont transmises par référence et non par valeur. Lorsque vous transférez une fonction en tant qu'argument, vous utilisez uniquement l'identifiant et non l'opérateur parenthèses qui permet d'appeler la méthode. Par exemple, le code suivant transfère une fonction appelée `clickListener()` en tant qu'argument à la méthode `addEventListener()` :

```
addEventListener(MouseEvent.CLICK, clickListener);
```

Même si cela peut sembler étrange pour les programmeurs découvrant ActionScript, les fonctions peuvent avoir des propriétés et des méthodes, comme les objets. Chaque fonction a en réalité une propriété en lecture seule appelée `length` qui stocke le nombre de paramètres définis pour la fonction. Ceci est différent de la propriété `arguments.length` qui indique le nombre d'arguments envoyés à la fonction. Dans ActionScript, le nombre d'arguments envoyés à une fonction peut dépasser le nombre de paramètres définis pour cette dernière. L'exemple suivant (qui compile uniquement en mode standard car le mode strict exige une correspondance exacte entre le nombre d'arguments transférés et le nombre de paramètres définis) illustre la différence entre les deux propriétés :

```
// Compiles only in standard mode
function traceLength(x:uint, y:uint):void
{
    trace("arguments received: " + arguments.length);
    trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

En mode standard, vous pouvez définir vos propriétés en dehors du corps de la fonction. Les propriétés de fonction peuvent servir de propriétés quasi statiques vous permettant de sauvegarder l'état d'une variable liée à la fonction. Par exemple, vous pouvez suivre le nombre d'appels d'une fonction particulière. Une telle fonctionnalité peut être utile si vous écrivez un jeu et souhaitez suivre le nombre de fois qu'un utilisateur se sert d'une certaine commande (vous pouvez également utiliser une propriété de classe statique). L'exemple suivant (qui compile uniquement en mode standard car le mode strict n'autorise pas l'ajout de propriétés dynamiques aux fonctions) crée une propriété de fonction en dehors de la déclaration de la fonction et incrémente cette propriété à chaque appel de la fonction :

```
// Compiles only in standard mode
var someFunction:Function = function ():void
{
    someFunction.counter++;
}

someFunction.counter = 0;

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

Domaine d'une fonction

Le domaine d'une fonction détermine non seulement l'endroit où cette fonction peut être appelée dans un programme, mais également les définitions auxquelles la fonction peut accéder. Les mêmes règles de domaine qui s'appliquent aux identifiants de variable s'appliquent aux identifiants de fonction. Une fonction déclarée dans le domaine global est disponible dans tout votre code. Par exemple, ActionScript 3.0 contient des fonctions globales (`isNaN()` et `parseInt()`, par exemple) disponibles n'importe où dans votre code. Une fonction imbriquée (une fonction déclarée dans une autre fonction) peut être utilisée n'importe où dans la fonction dans laquelle elle a été déclarée.

Chaîne de domaine

Chaque fois qu'une fonction commence une exécution, des objets et des propriétés sont créés. Premièrement, un objet spécial appelé *objet d'activation* est créé. Il stocke les paramètres et les variables locales ou fonctions déclarées dans le corps de la fonction. Vous ne pouvez pas accéder directement à l'objet d'activation car il s'agit d'un mécanisme interne. Deuxièmement, une *chaîne de domaine* est créée. Elle contient une liste ordonnée d'objets dans laquelle le moteur d'exécution recherche des déclarations d'identifiant. Chaque fonction qui s'exécute a une chaîne de domaine stockée dans une propriété interne. Dans le cas d'une fonction imbriquée, la chaîne de domaine commence avec son objet d'activation, suivi par l'objet d'activation de sa fonction parent. La chaîne continue de cette façon jusqu'à ce que l'objet global soit atteint. L'objet global est créé lorsqu'un programme ActionScript commence, et contient toutes les fonctions et les variables globales.

Fonctions closure

Une fonction *closure* est un objet qui contient un instantané d'une fonction et de son *environnement lexical*. L'environnement lexical d'une fonction comprend toutes les variables, propriétés, méthodes et les objets dans la chaîne de domaine de la fonction, ainsi que leurs valeurs. Les fonctions closure sont créées chaque fois qu'une fonction est exécutée à part d'un objet ou d'une classe. Le fait que les fonctions closure conservent le domaine dans lequel elles ont été définies crée des résultats intéressants lorsqu'une fonction est transférée en tant qu'argument ou valeur de renvoi dans un domaine différent.

Par exemple, le code suivant crée deux fonctions : `foo()`, qui renvoie une fonction imbriquée appelée `rectArea()` qui calcule la surface d'un rectangle, et `bar()`, qui appelle `foo()` et stocke la fonction closure renvoyée dans une variable appelée `myProduct`. Même si la fonction `bar()` définit sa propre variable locale `x` (avec une valeur de 2), lorsque la fonction closure `myProduct()` est appelée, elle conserve la variable `x` (avec une valeur de 40) définie dans la fonction `foo()`. La fonction `bar()` renvoie par conséquent la valeur `160` au lieu de `8`.

```
function foo():Function
{
    var x:int = 40;
    function rectArea(y:int):int // function closure defined
    {
        return x * y
    }
    return rectArea;
}
function bar():void
{
    var x:int = 2;
    var y:int = 4;
    var myProduct:Function = foo();
    trace(myProduct(4)); // function closure called
}
bar(); // 160
```

Les méthodes se comportent de la même façon car elles conservent également les informations concernant l'environnement lexical dans lequel elles ont été créées. Cette caractéristique se remarque plus particulièrement lorsqu'une méthode est extraite de son occurrence, ce qui crée une méthode liée. La différence principale entre une fonction closure et une méthode liée est que la valeur du mot-clé `this` dans une méthode liée se réfère toujours à l'occurrence à laquelle elle était associée à l'origine, alors que dans une fonction closure, la valeur du mot-clé `this` peut changer.

Chapitre 4 : Programmation orientée objets en ActionScript

Introduction à la programmation orientée objets

La programmation orientée objets (POO) est une technique d'organisation du code d'un programme en le groupant en objets. Les *objets* sont ici des éléments individuels comportant des informations (valeurs de données) et des fonctionnalités. L'approche orientée objet permet de regrouper des éléments particuliers d'informations avec des fonctionnalités ou des actions communes associées à ces informations. Vous pourriez, par exemple, regrouper les informations d'un enregistrement musical (titre de l'album, titre de la piste ou nom de l'artiste) avec des fonctionnalités comme l'ajout de la piste à une liste de lecture ou la lecture de tous les enregistrement de cet artiste. Ces éléments sont rassemblés en un seul élément, l'objet (par exemple, un « album » ou une « piste »). La possibilité d'intégrer ainsi toutes ces valeurs et ces fonctions offre divers avantages : il est possible de n'utiliser qu'une seule variable plutôt que plusieurs d'entre elles, ainsi que de regrouper des fonctionnalités liées entre elles. La combinaison des informations et des fonctionnalités permet de structurer les programmes pour qu'ils se rapprochent davantage du fonctionnement humain.

Classes

Une classe est une représentation abstraite d'un objet. Une classe conserve des informations sur les types de données contenues par un objet et sur les comportements possibles de cet objet. L'utilité de ce niveau d'abstraction peut ne pas être évidente dans le cas de petits scripts ne contenant que quelques objets destinés à interagir les uns avec les autres. Cependant, au fur et à mesure qu'un programme croît en ampleur, le nombre d'objets à gérer augmente. Les classes autorisent alors un meilleur contrôle sur la création des objets et sur leurs interactions.

Dès la première version d'ActionScript, les programmeurs en ActionScript pouvaient utiliser des objets Function pour créer des éléments ressemblant à des classes. ActionScript 2.0 a ensuite ajouté une prise en charge formelle des classes, avec des mots-clés tels que `class` et `extends`. De son côté, ActionScript 3.0 préserve la prise en charge des mots-clés introduits avec ActionScript 2.0, tout en ajoutant de nouvelles possibilités : par exemple un meilleur contrôle d'accès avec les attribut `protected` et `internal`, ainsi qu'un meilleur contrôle de l'héritage avec les mots-clés `final` et `override`.

Si vous avez déjà créé des classes dans des langages de programmation tels que Java, C++ ou C#, vous ne serez pas dépaysé par ActionScript. ActionScript partage avec ces langages de nombreux mots-clés et noms d'attributs, comme `classe`, `extends` et `public`.

Remarque : *dans la documentation Adobe ActionScript, le terme propriété désigne tout membre d'un objet ou d'une classe (variables, constantes et méthodes). En outre, bien que les termes classe et statique soient fréquemment utilisés indifféremment, nous ferons une distinction entre ces termes. Par exemple, l'expression « propriétés de classe » désigne tous les membres d'une classe plutôt que ses membres statiques exclusivement.*

Définitions de classe

En ActionScript 3.0, les définitions de classe utilisent la même syntaxe qu'en ActionScript 2.0. La syntaxe correcte d'une définition de classe utilise le mot-clé `class` suivi du nom de la classe. Le corps de la définition de classe est inséré entre des accolades ({}) après le nom de la classe. Par exemple, le code suivant crée une classe appelée Shape qui contient une variable appelée `visible` :

```
public class Shape
{
    var visible:Boolean = true;
}
```

Notez que la syntaxe est différente dans le cas des définitions de classe faisant partie d'un package. En ActionScript 2.0, si une classe fait partie d'un package, le nom de ce dernier doit figurer dans la déclaration de classe. Comme l'instruction `package` a été introduite en ActionScript 3.0, le nom du package doit figurer dans la déclaration de package et non pas dans la déclaration de classe. Par exemple, les déclarations de classe suivantes montrent comment la classe BitmapData, qui fait partie du package flash.display, est définie dans ActionScript 2.0 et ActionScript 3.0 :

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

Attributs de classe

ActionScript 3.0 vous permet de modifier des définitions de classe à l'aide de l'un des quatre attributs suivants :

Attribut	Définition
`dynamique`	Permet d'ajouter des propriétés aux occurrences lors de l'exécution.
`final`	Ne doit pas être étendue par une autre classe.
`interne` (par défaut)	Visible pour les références à partir du package actuel.
`public`	Visible pour les références à partir de n'importe quel point du code.

Pour chacun de ces attributs, à l'exception d'`internal`, vous incluez explicitement l'attribut pour obtenir le comportement qui lui est associé. Par exemple, faute d'inclure l'attribut `dynamic` lors de la définition d'une classe, vous n'êtes pas en mesure d'ajouter des propriétés à une occurrence de classe lors de l'exécution. Pour affecter explicitement un attribut, placez-le au début de la définition de la classe, comme dans le code ci-dessous :

```
dynamic class Shape {}
```

Notez que la liste ne contient pas d'attribut appelé `abstract`. En effet, les classes abstraites ne sont pas prises en charge en ActionScript 3.0. Notez également que la liste ne comprend pas non plus les attributs `private` et `protected`. Ces attributs n'ont de signification qu'au sein d'une définition de classe et ne peuvent être appliqués aux classes elles-mêmes. Si vous ne souhaitez pas qu'une classe soit visible hors de son package, placez-la au sein d'un package et affectez la classe de l'attribut `internal`. Autrement, vous pouvez omettre les attributs `internal` et `public` et le compilateur ajoutera automatiquement l'attribut `internal` pour vous. Vous pouvez également définir une classe afin qu'elle soit uniquement visible à l'intérieur du fichier source dans lequel elle est définie. Pour ce faire, placez-la à la fin de ce fichier source, après l'accolade de fin de la définition du package.

Corps de la classe

Le corps de la classe, qui est entouré d'accolades, définit les variables, les constantes et les méthodes de la classe. L'exemple suivant illustre la déclaration de la classe Accessibility dans ActionScript 3.0 :

```
public final class Accessibility
{
    public static function get active():Boolean;
    public static function updateProperties():void;
}
```

Vous pouvez aussi définir un espace de noms au sein d'un corps de classe. L'exemple suivant montre la définition d'un espace de noms dans le corps d'une classe et son utilisation comme attribut d'une méthode de cette classe :

```
public class SampleClass
{
    public namespace sampleNamespace;
    sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 vous permet d'inclure dans un corps de classe non seulement des définitions, mais également des instructions. Les instructions qui figurent dans le corps d'une classe, mais hors d'une définition de méthode, sont exécutées une seule fois, lors de la première apparition de la définition de classe et de la création de l'objet class qui lui est associé. L'exemple suivant comprend un appel vers une fonction externe, `hello()` et une instruction `trace` qui affiche un message de confirmation lorsque la classe est définie.

```
function hello():String
{
    trace("hola");
}
class SampleClass
{
    hello();
    trace("class created");
}
// output when class is created
hola
class created
```

En ActionScript 3.0, il est permis de définir une propriété statique et une propriété d'occurrence portant le même nom dans le corps de la même classe. Par exemple, le code suivant déclare une variable statique appelée `message` et une variable d'occurrence du même nom :

```
class StaticTest
{
    static var message:String = "static variable";
    var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // output: static variable
trace(myST.message); // output: instance variable
```

Attributs de propriété de classe

Dans le cadre du modèle d'objet ActionScript, le terme *propriété* représente tout ce qui peut être membre d'une classe : variables, constantes et méthodes. Ce terme est utilisé de manière plus restrictive dans le manuel Guide de référence du langage et des composants ActionScript 3.0 : il ne désigne alors que les membres d'une classe qui sont des variables ou qui sont définis par une méthode de lecture/définition. En ActionScript 3.0, il existe un jeu d'attributs qui peut être utilisé avec n'importe quelle propriété de classe. Le tableau suivant présente ce jeu d'attributs.

Attribut	Définition
`internal` (par défaut)	Visible pour les références au sein d'un même package.
`private`	Visible pour les références au sein d'une même classe.
`protected`	Visible pour des références au sein d'une même classe et de classes dérivées.
`public`	Visible pour des références en tous lieux.
`static`	Spécifie qu'une propriété appartient à la classe et non pas aux occurrences de la classe.
`UserDefinedNamespace`	Nom d'espace de noms défini par l'utilisateur.

Attributs d'espace de noms pour le contrôle d'accès

ActionScript 3.0 comporte quatre attributs spéciaux qui contrôlent l'accès aux propriétés définies au sein d'une classe : `public`, `private`, `protected` et `internal`.

Avec l'attribut `public`, une propriété est visible de n'importe quel point du script. Par exemple, si vous souhaitez qu'une méthode soit disponible pour du code externe au package, vous devez la déclarer avec l'attribut `public`. Ceci est valable pour toute propriété, qu'elle soit déclarée à l'aide des mots-clés `var`, `const` ou `function`.

Avec l'attribut `private`, une propriété n'est visible qu'à partir de la classe où cette propriété est définie. Ce comportement est différent de celui de l'attribut `privé` en ActionScript 2.0, où une sous-classe pouvait accéder à une propriété déclarée private d'une superclasse. L'accès en cours d'exécution constitue aussi un changement radical de comportement. En ActionScript 2.0, la restriction d'accès introduite par le mot-clé `private` ne portait que sur la compilation et il était facile de la contourner lors de l'exécution. Cette situation n'existe plus en ActionScript 3.0. Les propriétés désignées comme `private` sont indisponibles, aussi bien au cours de la compilation que de l'exécution.

Par exemple, le code ci-dessous crée une classe simple appelée PrivateExample pourvue d'une variable privée. Elle tente ensuite d'accéder à cette variable depuis un emplacement hors de la classe.

```
class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in ActionScript 3.0, this
is a run-time error.
```

En ActionScript 3.0, toute tentative d'accéder à une propriété private à l'aide de l'opérateur point (`myExample.privVar`) déclenche une erreur de compilation en mode strict. Autrement, l'erreur est signalée à l'exécution, tout comme lors de l'utilisation de l'opérateur d'accès à la propriété (`myExample["privVar"]`).

Le tableau suivant présente les divers résultats d'une tentative d'accès à une propriété déclarée comme privée qui appartient à une classe scellée (non dynamique) :

	Mode strict	**Mode standard**
opérateur point (`.`)	erreur à la compilation	erreur à l'exécution
opérateur crochet (`[]`)	erreur à l'exécution	erreur à l'exécution

Dans les classes déclarées avec un attribut `dynamic`, les tentatives d'accès à une variable privée ne provoquent pas d'erreur d'exécution. La variable n'est simplement pas visible, de sorte que la valeur `undefined` est renvoyée. Une erreur à la compilation survient néanmoins si vous utilisez l'opérateur point en mode strict. L'exemple suivant est identique au précédent si ce n'est que la classe PrivateExample est déclarée en tant que classe dynamique :

```
dynamic class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // output: undefined
```

Les classes dynamiques renvoient le plus souvent la valeur `undefined` plutôt que de générer une erreur lorsque du code, extérieur à une classe, tente d'accéder à une classe déclarée comme propriété privée. Le tableau suivant montre qu'une erreur est générée uniquement lorsque l'opérateur point est utilisé pour accéder à une propriété privée en mode strict :

	Mode strict	**Mode standard**
opérateur point (`.`)	erreur à la compilation	`undefined`
opérateur crochet (`[]`)	`undefined`	`undefined`

Avec l'attribut `protected`, nouveau en ActionScript 3.0, une propriété n'est visible qu'à partir de sa propre classe ou d'une sous-classe de celle-ci. Autrement dit, une propriété déclarée protected n'est disponible qu'à partir de sa propre classe ou des classes qui lui sont inférieures dans sa hiérarchie d'héritage, que la sous-classe se trouve dans le même package ou dans un autre.

Pour les programmeurs qui connaissent ActionScript 2.0, cette fonctionnalité est semblable à l'attribut `private` de ce langage. L'attribut `protected` d'ActionScript 3.0 est également semblable à l'attribut `protected` de Java, à la différence près que la version de Java autorise également l'accès à partir d'un même package. L'attribut `protected` est utile pour créer une variable ou une méthode nécessaire aux sous-classes, mais qui ne doit pas être visible à partir du code extérieur à la hiérarchie d'héritage.

Avec l'attribut `interne` qui apparaît avec ActionScript 3.0, une propriété n'est visible qu'à partir de son propre package. Il s'agit de l'attribut par défaut pour du code dans un package et il s'applique à toute propriété dépourvue de l'un quelconque des attributs suivants :

- `public`
- `private`
- `protected`
- un espace de noms défini par l'utilisateur

L'attribut `internal` est semblable au contrôle d'accès par défaut de Java quoiqu'en Java il n'existe pas de nom explicite pour ce niveau d'accès ; il ne peut être obtenu que par l'omission de tout autre modificateur d'accès. Avec l'attribut `internal`, qui apparaît avec ActionScript 3.0, il est possible d'indiquer explicitement votre intention de ne rendre une propriété visible qu'à partir de son propre package.

Attribut static

L'attribut `static`, qui peut être utilisé avec des propriétés déclarées à l'aide des mots-clés `var`, `const` ou `function`, vous permet d'associer une propriété à la classe elle-même plutôt qu'à ses occurrences. Le code externe à cette classe doit appeler les propriétés statiques à l'aide du nom de la classe et non pas à partir du nom d'une occurrence.

Les sous-classes n'héritent pas des propriétés statiques ; ces dernières font partie de leur chaîne de portée. En d'autres termes, dans le corps d'une sous-classe, une variable ou une méthode statique peut être utilisée sans faire référence à la classe dans laquelle elle a été définie.

Attributs d'espace de noms définis par l'utilisateur

Comme solution de rechange aux attributs de contrôle d'accès prédéfinis, vous pouvez créer un espace de noms personnalisé pour l'utiliser comme attribut. Il ne peut exister qu'un seul attribut d'espace de noms par définition et vous ne pouvez pas associer cet attribut à tout attribut de contrôle d'accès (`public`, `private`, `protected`, `internal`).

Variables

Pour déclarer une variable, utilisez les mots-clés `var` ou `const`. Les variables déclarées avec le mot-clé `var` sont susceptibles de changer de valeur plusieurs fois au cours de l'exécution d'un script. Les variables déclarées à l'aide du mot-clé `const` sont appelées des *constantes* et on ne peut leur attribuer une valeur qu'une seule fois. Une erreur survient lors d'une tentative d'attribution d'une nouvelle valeur à une constante initialisée.

Variables statiques

Les variables statiques sont déclarées à l'aide de l'association du mot-clé `static` et de l'instruction `var` ou `const`. Les variables statiques sont affectées à une classe, plutôt qu'à une occurrence de classe. Elles permettent de stocker et de partager des informations propres à une classe entière d'objets. Par exemple, une variable statique permet d'enregistrer le nombre de fois où une classe a été instanciée ou bien le nombre maximal d'occurrences autorisées pour une classe.

L'exemple ci-dessous crée une variable `totalCount` qui permet d'enregistrer le nombre total d'instanciations d'une classe et une constante `MAX_NUM` dont la valeur est le nombre maximum d'instanciations autorisées. Les variables `totalCount` et `MAX_NUM` sont statiques car elles contiennent des valeurs qui s'appliquent à la classe elle-même, plutôt qu'à une occurrence particulière.

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

Un code extérieur à la classe StaticVars, ainsi que toutes ses sous-classes, ne peuvent faire référence aux propriétés `totalCount` et `MAX_NUM` que par le biais de la classe elle-même. Par exemple, le code suivant fonctionne :

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

Comme il est impossible d'accéder à des variables statiques par une occurrence de la classe, le code suivant renvoie des erreurs :

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

Les variables qui sont déclarées avec les deux mots-clés `static` et `const` doivent être initialisées en même temps lors de la déclaration de la constante, tout comme la classe StaticVars le fait pour `MAX_NUM`. Il est impossible d'attribuer une valeur à `MAX_NUM` au sein du constructeur ou d'une méthode d'occurrence. Le code suivant génère une erreur, car ce n'est pas une façon valide d'initialiser une constante statique :

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

Variables d'occurrence

Les variables d'occurrence contiennent des propriétés déclarées à l'aide des mots-clés `var` et `const`, mais sans le mot-clé `static`. Les variables d'occurrence, qui sont associées à des occurrences de classe plutôt qu'à la classe elle-même, sont utiles pour conserver des valeurs spécifiques à une occurrence. Par exemple, la classe Array dispose d'une propriété d'occurrence appelée `length` qui conserve le nombre d'éléments du tableau appartenant à une occurrence particulière de la classe Array.

Qu'elles soient déclarées avec le mot-clé `var` ou `const`, les variables d'occurrence ne peuvent pas être redéfinies dans une sous-classe. Il est toutefois possible d'obtenir un effet similaire à la redéfinition de variables, en redéfinissant des méthodes de lecture et de définition.

Méthodes

Les méthodes sont des fonctions associées à la définition d'une classe. Dès la création d'une occurrence de la classe, une méthode est liée à cette occurrence. Contrairement à une fonction déclarée hors d'une classe, une méthode ne peut pas être utilisée séparément de l'occurrence à laquelle elle est associée.

Les méthodes sont définies à l'aide du mot-clé `function`. Comme toute propriété de classe, vous pouvez appliquer n'importe quel attribut de propriété de classe aux méthodes, qu'elles soient privées, protégées, publiques, internes ou statiques, ainsi qu'à un espace de noms personnalisé. Vous pouvez utiliser une instruction function telle que :

```
public function sampleFunction():String {}
```

Vous pouvez aussi utiliser une variable à laquelle vous attribuez une expression de fonction, comme ci-dessous :

```
public var sampleFunction:Function = function () {}
```

Dans la plupart des cas, il est préférable d'utiliser une instruction function plutôt qu'une expression de fonction pour les raisons suivantes :

- Les instructions function sont plus concises et plus faciles à lire.
- Elles vous permettent d'utiliser les mots-clés `override` et `final`.
- Les instructions function créent une liaison plus robuste entre l'identifiant (le nom de la fonction) et le code dans le corps de la méthode. Comme la valeur d'une variable peut être modifiée par une instruction assignment, le lien entre une variable et son expression de fonction peut être rompu à tout moment. Bien qu'il soit possible de résoudre ce problème en déclarant la variable avec `const` au lieu de `var`, cette technique n'est pas recommandée car elle rend le code difficilement lisible et empêche d'utiliser les mots-clés `override` et `final`.

Il existe toutefois un cas dans lequel une expression de fonction doit être utilisée : si vous choisissez d'affecter une fonction à l'objet prototype.

Méthodes constructeur

Les méthodes constructeur, parfois appelées simplement *constructeurs*, sont des fonctions qui portent le nom de la classe dans laquelle elles sont définies. Tout code qui figure dans une méthode constructeur est exécuté toutes les fois qu'une occurrence de la classe est créée à l'aide du mot-clé `new`. Par exemple, le code suivant définit une classe simple appelée Example qui contient une propriété unique appelée `status`. La valeur initiale de la variable `status` est fixée dans la fonction constructeur.

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

Les méthodes constructeur ne peuvent être que publiques, mais l'utilisation de l'attribut `public` est facultative. Il est impossible d'utiliser l'un des autres spécificateurs de contrôle d'accès, y compris `private`, `protected` ou `internal` avec un constructeur. De même qu'il est impossible d'utiliser non plus, avec un constructeur, un espace de noms défini par l'utilisateur.

Un constructeur peut appeler explicitement le constructeur de sa superclasse directe, à l'aide de l'instruction `super()`. Si le constructeur de la superclasse n'est pas explicitement appelé, le compilateur insère automatiquement un appel devant la première instruction dans le corps du constructeur. Vous pouvez aussi appeler des méthodes de la superclasse à l'aide du préfixe `super` en référence à la superclasse. Si vous décidez d'utiliser à la fois `super()` et `super` dans le corps du même constructeur, veillez à appeler `super()` en premier. Sinon, la référence `super` n'a pas le comportement prévu. Le constructeur `super()` devrait également être appelé avant toute instruction `throw` ou `return`.

L'exemple suivant décrit ce qui se passe si vous tentez d'utiliser la référence `super` avant d'appeler le constructeur `super()`. Une nouvelle classe, ExampleEx, étend la classe Example. Le constructeur ExampleEx tente d'accéder à la variable d'état définie dans sa super classe, mais avant un appel à `super()`. L'instruction `trace()` du constructeur ExampleEx produit la valeur `null` car la variable `status` n'est pas disponible tant que le constructeur `super()` n'a pas été exécuté.

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

Bien que l'utilisation de l'instruction `return` au sein d'un constructeur soit autorisée, il n'est pas permis de renvoyer une valeur. En d'autres termes, aucune expression ou valeur ne peut être associée à l'instruction `return`. Par conséquent, les méthodes constructeur ne sont pas autorisées à renvoyer des valeurs, ce qui signifie qu'aucun type de valeur renvoyée ne peut être spécifié.

Si vous ne définissez pas de méthode constructeur dans la classe, le compilateur crée automatiquement un constructeur vide. Si la classe en étend une autre, le compilateur insère un appel `super()` dans le constructeur qu'il génère.

Méthodes statiques

Les méthodes statiques, également appelées parfois *méthodes de classe*, sont déclarées avec le mot-clé `static`. Les méthodes statiques sont affectées à une classe plutôt qu'à une occurrence de classe. Elles permettent d'encapsuler des fonctionnalités qui ont une portée plus étendue que l'état d'une occurrence individuelle. Comme les méthodes statiques sont associées à l'intégralité d'une classe, on peut accéder à des méthodes statiques uniquement par une classe et pas du tout par une occurrence de classe.

Les méthodes statiques permettent d'encapsuler des fonctionnalités qui ne se bornent pas à la modification d'état des occurrences de classe. Autrement dit, une méthode devrait être statique si elle offre des fonctionnalités qui n'affectent pas directement la valeur d'une occurrence de classe. Par exemple, la classe Date possède une méthode statique appelée `parse()` qui reçoit une chaîne et la convertit en nombre. La méthode est statique parce qu'elle n'affecte pas une occurrence individuelle de sa classe. La méthode `parse()`, reçoit une chaîne représentant une valeur de date, l'analyse et renvoie un nombre dans un format compatible avec la représentation interne d'un objet Date. Cette méthode n'est pas une méthode d'occurrence, puisqu'il n'y aurait aucun intérêt à l'appliquer à une occurrence de la classe Date.

Comparons la méthode statique `parse()` à l'une des méthodes d'occurrence de la classe Date, comme `getMonth()`. La méthode `getMonth()` est une méthode d'occurrence parce qu'elle agit directement sur la valeur d'une occurrence en récupérant un composant spécifique, le mois, d'une occurrence de Date.

Comme les méthodes statiques ne sont pas liées à des occurrences individuelles, vous ne pouvez pas utiliser les mots-clés `this` ou `super` dans le corps d'une méthode statique. Les deux références `this` et `super` n'ont de sens que dans le contexte d'une méthode d'occurrence.

Contrairement à d'autres langages de programmation basés sur des classes, les méthodes statiques en ActionScript 3.0 ne sont pas héritées.

Méthodes d'occurrence

Les méthodes d'occurrence sont déclarées sans le mot-clé `static`. Les méthodes d'occurrence, qui sont affectées aux occurrences d'une classe et non pas à la classe elle-même, permettent d'implémenter des fonctionnalités qui affectent des occurrences individuelles d'une classe. Par exemple, la classe Array contient une méthode d'occurrence appelée `sort()` qui opère directement sur des occurrences Array.

Dans le corps d'une méthode d'occurrence, les variables statiques et d'occurrence sont de même portée ; ce qui signifie que les variables définies dans la même classe peuvent être référencées à l'aide d'un identificateur simple. Par exemple, la classe suivante, CustomArray, étend la classe Array. La classe CustomArray définit une variable statique appelée `arrayCountTotal` destinée à contenir le nombre total d'occurrences de la classe, une variable d'occurrence appelée `arrayNumber` qui enregistre l'ordre dans lequel les occurrences ont été créées et une méthode d'occurrence appelée `getPosition()` qui renvoie les valeurs de ces variables.

```
public class CustomArray extends Array
{
    public static var arrayCountTotal:int = 0;
    public var arrayNumber:int;

    public function CustomArray()
    {
        arrayNumber = ++arrayCountTotal;
    }

    public function getArrayPosition():String
    {
         return ("Array " + arrayNumber + " of " + arrayCountTotal);
    }
}
```

Pour accéder à la variable statique `arrayCountTotal`, du code externe à cette classe doit passer par l'objet class utilisant `CustomArray.arrayCountTotal` ; mais le code qui réside dans le corps de la méthode `getPosition()` peut faire référence directement à la variable statique `arrayCountTotal`. C'est également le cas pour les variables statiques dans les superclasses. Bien que les propriétés statiques ne soient pas héritées en ActionScript 3.0, les propriétés statiques des superclasses sont dans la portée. Par exemple, la classe Array possède quelques variables statiques, dont l'une est une constante appelée `DESCENDING`. Le code qui réside dans une sous-classe d'Array peut accéder à la constante statique `DESCENDING` à l'aide d'un identifiant simple :

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

La valeur de la référence `this` dans le corps d'une méthode d'occurrence est une référence à l'occurrence à laquelle la méthode est affectée. Le code suivant montre que la référence `this` pointe sur l'occurrence qui contient la méthode :

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

Il est possible de contrôler l'héritage des méthodes d'occurrence à l'aide des mots-clés `override` et `final`. Vous pouvez utiliser l'attribut `override` pour redéfinir une méthode héritée et l'attribut `final` pour empêcher les sous-classes de redéfinir une méthode.

Méthodes accesseur get et set

Les fonctions d'accesseur de lecture et de définition, appelées aussi accesseurs *Get* et *Set*, vous permettent de suivre les principes de programmation sur le masquage et l'encapsulation des informations tout en offrant une interface de programmation conviviale pour les classes que vous créez. Les fonctions de lecture et de définition (get et set) permettent de garder privées les propriétés d'une classe. Par contre, elles permettent à des utilisateurs de votre classe d'accéder à ces propriétés comme s'ils accédaient à une variable de classe au lieu d'appeler une méthode de classe.

L'avantage de cette approche est qu'elle permet d'éviter les fonctions d'accesseur traditionnelles aux noms compliqués, comme `getPropertyName()` et `setPropertyName()`. Leur autre avantage est qu'elles évitent d'avoir deux fonctions exposées publiquement pour chaque propriété accessible en lecture et en écriture.

Dans l'exemple suivant, la classe appelée GetSet, possède des fonctions accesseurs de lecture et de définition appelées `publicAccess()` qui permettent d'accéder à la variable privée appelée `privateProperty` :

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

Si vous tentez d'accéder directement à la propriété `privateProperty`, une erreur se produit :

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

En revanche, si vous utilisez la classe GetSet, vous faites appel à ce qui paraît être une propriété appelée `publicAccess`, mais qui correspond, en fait, à une paire de fonctions accesseurs de lecture et de définition intervenant sur la propriété privée appelée `privateProperty`. L'exemple suivant instancie la classe GetSet, puis définit la valeur de la propriété `privateProperty` à l'aide de l'accesseur public appelé `publicAccess` :

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

Les fonctions d'accesseur Get et Set permettent également de redéfinir des propriétés héritées d'une superclasse, ce qui n'est pas possible avec des variables régulières membres de classes. Les variables des membres de la classe qui sont déclarées à l'aide du mot-clé `var` ne peuvent pas être redéfinies dans une sous-classe. Toutefois, cette restriction ne concerne pas les propriétés créées à l'aide des fonctions d'accesseur Get et Set. Vous pouvez utiliser l'attribut `override` sur des fonctions d'accesseur Get et Set héritées d'une superclasse.

Méthodes liées

Une méthode liée, parfois appelée *fermeture de méthode*, est tout simplement une méthode extraite de son occurrence. On peut citer comme exemples les méthodes passées en arguments à une fonction ou renvoyées comme valeurs par une fonction. La méthode liée, qui est une nouveauté d'ActionScript 3.0, est semblable à une fermeture de fonction dans la mesure où elle conserve son environnement lexical, même après avoir été extraite de son occurrence. Toutefois, la différence entre une méthode liée et une fermeture de fonction réside dans le fait que la référence `this` d'une méthode liée reste liée à l'occurrence qui implémente cette méthode. Autrement dit, la référence `this` d'une méthode liée pointe toujours sur l'objet original qui a implémenté la méthode. Pour les fermetures de fonction, la référence `this` est générique, ce qui signifie qu'elle pointe sur l'objet auquel est associée la fonction lorsqu'elle est appelée.

Il est important de comprendre les méthodes liées pour utiliser le mot-clé `this` à bon escient. N'oubliez pas que `this` représente une référence à l'objet parent d'une méthode. La plupart des programmeurs en ActionScript s'attendent à ce que le mot-clé `this` représente toujours l'objet ou la classe qui contient la définition d'une méthode. Ce n'est pas toujours le cas sans méthode liée. Par exemple, dans les versions précédentes d'ActionScript, la référence `this` ne pointait pas toujours sur l'occurrence qui implémentait la méthode. En ActionScript 2.0, lorsque les méthodes sont extraites d'une occurrence, non seulement la référence `this` n'est pas liée à l'occurrence originale, mais les variables et les méthodes de la classe de cette occurrence ne sont pas disponibles. Toutefois, ce problème n'existe plus avec ActionScript 3.0, car les méthodes liées sont automatiquement créées lorsque la méthode est passée en paramètre. Avec les méthodes liées, le mot-clé `this` fait toujours référence à l'objet ou à la classe dans laquelle la méthode est définie.

Le code suivant définit une classe appelée ThisTest, qui contient une méthode appelée `foo()` définissant la méthode liée et une méthode appelée `bar()` qui renvoie cette méthode liée. Le code extérieur à la classe crée une occurrence de la classe ThisTest, appelle la méthode `bar()` et enregistre la valeur à renvoyer dans la variable `myFunc`.

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

Les deux dernières lignes de code montrent que la référence `this` dans la méthode liée `foo()` pointe encore sur une occurrence de la classe ThisTest, bien que la référence `this` de la ligne précédente pointe sur l'objet global. De plus, la méthode liée, stockée dans la variable `myFunc`, peut encore accéder aux variables membres de la classe ThisTest. Si ce code est exécuté en ActionScript 2.0, les références `this` seront identiques et la variable `num` sera `undefined`.

Les gestionnaires d'événement constituent un domaine dans lequel l'ajout des méthodes liées est le plus notable, car la méthode `addEventListener()` nécessite de transmettre une fonction ou une méthode en argument.

Enumérations et classes

Les *énumérations* sont des types de données que vous pouvez créer pour encapsuler un petit ensemble de valeur. Contrairement à C++ avec le mot-clé `enum` et à Java avec l'interface d'énumération, ActionScript 3.0 ne dispose pas d'un mécanisme d'énumération spécifique. Il est toutefois possible de créer des énumérations à l'aide de classes et de constantes statiques. Par exemple, en ActionScript 3.0, la classe PrintJob utilise une énumération appelée PrintJobOrientation pour stocker les valeurs `"landscape"` et `"portrait"`, comme l'indique le code ci-dessous :

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

Par convention, une classe d'énumération est déclarée avec l'attribut `final`, car il n'est pas nécessaire d'étendre cette classe. Cette classe étant composée uniquement de membres statiques, il est impossible d'en créer des occurrences. En effet, on accède aux valeurs de l'énumération directement par l'objet classe, comme le montre l'extrait de code suivant :

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

Toutes les classes d'énumération d'ActionScript 3.0 contiennent uniquement des variables de type String, int ou uint. Le fait que les fautes de frappe sont plus faciles à détecter avec les énumérations présente un grand avantage par rapport à des chaînes littérales ou des nombres. Si vous faites une erreur dans le nom d'une énumération, le compilateur ActionScript génère une erreur. Si vous utilisez des valeurs littérales, le compilateur acceptera un nom mal épelé ou un chiffre erroné. Dans l'exemple ci-dessus, le compilateur génère une erreur si le nom de la constante d'énumération est incorrect, comme dans l'extrait suivant :

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

Toutefois, le compilateur ne génère pas d'erreur si vous faites une faute de frappe dans le nom d'une chaîne littérale :

```
    if (pj.orientation == "portrai") // no compiler error
```

Une autre technique de création d'énumérations consiste à créer une classe séparée avec des propriétés statiques pour l'énumération. Toutefois, cette technique est différente dans la mesure où chacune des propriétés statiques contient une occurrence de la classe plutôt qu'une valeur chaîne ou un entier. Par exemple, le code suivant crée une classe d'énumération pour les jours de la semaine :

```
public final class Day
{
    public static const MONDAY:Day = new Day();
    public static const TUESDAY:Day = new Day();
    public static const WEDNESDAY:Day = new Day();
    public static const THURSDAY:Day = new Day();
    public static const FRIDAY:Day = new Day();
    public static const SATURDAY:Day = new Day();
    public static const SUNDAY:Day = new Day();
}
```

Cette technique n'est pas utilisée par ActionScript 3.0, mais de nombreux développeurs y ont recours car elle permet d'obtenir un meilleur type de vérification. Par exemple, une méthode qui renvoie une valeur d'énumération peut restreindre la valeur renvoyée au type de données de l'énumération. Le code suivant illustre non seulement une fonction qui renvoie un jour de la semaine, mais aussi un appel de fonction qui utilise le type énumération comme annotation de type :

```
function getDay():Day
{
    var date:Date = new Date();
    var retDay:Day;
    switch (date.day)
    {
        case 0:
            retDay = Day.MONDAY;
            break;
        case 1:
            retDay = Day.TUESDAY;
            break;
        case 2:
            retDay = Day.WEDNESDAY;
            break;
        case 3:
            retDay = Day.THURSDAY;
            break;
        case 4:
            retDay = Day.FRIDAY;
            break;
        case 5:
            retDay = Day.SATURDAY;
            break;
        case 6:
            retDay = Day.SUNDAY;
            break;
    }
    return retDay;
}

var dayOfWeek:Day = getDay();
```

Il est également possible d'améliorer la classe Day afin qu'elle associe un entier à chaque jour de la semaine et comporte une méthode `toString()` renvoyant une représentation du jour sous forme de chaîne.

Classes des éléments incorporés

ActionScript 3.0 utilise des classes spéciales, appelées *classes des éléments incorporés*, pour représenter les actifs incorporés. Un *élément incorporé* est un élément (son, image ou police) qui est incorporé au fichier SWF lors de la compilation. Contrairement au chargement dynamique, l'incorporation des actifs les rend disponibles immédiatement lors de l'exécution ; mais cette méthode augmente la taille des fichiers SWF.

Utilisation de classes d'actifs incorporés dans Flash Professional

Pour incorporer un élément, placez-le d'abord dans la bibliothèque d'un fichier FLA. Utilisez ensuite la propriété de liaison de l'élément pour fournir un nom à la classe de l'élément incorporé. S'il n'existe pas de classe de ce nom dans le chemin de classe indiqué, une classe est automatiquement créée. Vous pouvez alors créer une occurrence de la classe d'éléments incorporés et utiliser les propriétés et méthodes définies ou héritées par cette classe. Par exemple, le code suivant permet de lire un son intégré et lié à une classe d'éléments incorporés appelée PianoMusic :

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

Pour incorporer des actifs dans un projet Flash Professional, vous pouvez aussi utiliser la balise de métadonnées `[Embed]` (voir ci-après). Dans ce cas, Flash Professional fait appel au compilateur Flex, et non à son propre compilateur, pour compiler le projet.

Utilisation de classe d'actifs intégrés à l'aide du compilateur Flex

Lorsque vous utilisez le compilateur Flex, servez-vous de la balise de métadonnées `[Embed]` pour incorporer un actif dans le code ActionScript. Placez l'actif dans le dossier source principal ou dans un autre dossier qui se trouve dans le chemin de création. Lorsque le compilateur Flex trouve une balise de métadonnées Embed, il crée la classe d'actifs intégrés pour vous. Il est possible d'accéder à la classe par une variable de type de données Class que l'on déclare immédiatement après la balise de métadonnées `[Embed]`. Par exemple, le code ci-dessous intègre un son appelé sound1.mp3. Il utilise une variable appelée `soundCls` pour enregistrer une référence à la classe de l'élément intégré associé à ce son. L'exemple crée alors une occurrence de la classe de l'élément intégré et appelle la méthode `play()` sur cette occurrence :

```
package
{
    import flash.display.Sprite;
    import flash.media.SoundChannel;
    import mx.core.SoundAsset;

    public class SoundAssetExample extends Sprite
    {
        [Embed(source="sound1.mp3")]
        public var soundCls:Class;

        public function SoundAssetExample()
        {
            var mySound:SoundAsset = new soundCls() as SoundAsset;
            var sndChannel:SoundChannel = mySound.play();
        }
    }
}
```

Adobe Flash Builder

Pour utiliser la balise de métadonnées `[Embed]` dans un projet ActionScript Flash Builder, importez les classes nécessaires depuis la structure Flex. Par exemple, pour incorporer des sons, importez la classe mx.core.SoundAsset. Pour utiliser la structure Flex, ajoutez le fichier framework.swc à votre chemin de création ActionScript. La taille du fichier SWF va alors augmenter.

Adobe Flex

Dans Flex, vous pouvez aussi intégrer un actif à l'aide de la directive `@Embed()` dans une définition de balise MXML.

Interfaces

Une interface est une collection de déclarations de méthodes qui autorise les communications entre des objets différents. Par exemple, ActionScript 3.0 définit l'interface IEventDispatcher qui contient les déclarations des méthodes qu'une classe peut utiliser pour gérer les objets événements. L'interface IEventDispatcher établit une technique standard permettant aux objets de s'échanger les événements. Le code suivant représente la définition de l'interface IEventDispatcher :

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
            useCapture:Boolean=false, priority:int=0,
            useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
            useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

Les interfaces sont basées sur la distinction entre l'interface d'une méthode et l'implémentation de celle-ci. L'interface d'une méthode comprend toutes les informations nécessaires pour appeler cette méthode : le nom de la méthode, l'ensemble des paramètres qu'elle reçoit et le type de données qu'elle renvoie. L'implémentation d'une méthode comprend non seulement les informations de l'interface, mais aussi les instructions exécutables qui caractérisent le comportement de la méthode. La définition d'une interface ne contient que les interfaces de la méthode et toute classe qui implémente l'interface doit donc définir les implémentations de la méthode.

Dans ActionScript 3.0, la classe EventDispatcher implémente l'interface IEventDispatcher en définissant toutes les méthodes de cette interface et en ajoutant le corps de chacune de ces méthodes. Le code suivant est extrait de la définition de la classe EventDispatcher :

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

L'interface IEventDispatcher fait office de protocole utilisé par les occurrences d'EventDispatcher pour traiter les objets événements et les transmettre aux autres objets qui ont également implémenté cette interface.

Il est aussi possible de décrire une interface en disant qu'elle définit un type de données, au même titre qu'une classe. En conséquence, une interface peut être utilisée comme annotation de type, tout comme une classe. En tant que type de données, une interface peut également être utilisée avec des opérateurs, par exemple les opérateurs `is` et `as`, qui nécessitent un type de données. Toutefois, à l'inverse d'une classe, il n'est pas possible d'instancier une interface. C'est en raison de cette distinction que de nombreux programmeurs voient les interfaces comme des types de données abstraites et les classes comme des types de données concrètes.

Définition d'une interface

La structure de la définition d'une interface est similaire à celle de la définition d'une classe, à ceci près qu'une interface ne peut pas contenir les corps des méthodes. Les interfaces ne peuvent pas comporter des variables ou des constantes, mais elles peuvent contenir des méthodes de lecture et de définition. Pour définir une interface, on utilise le mot-clé `interface`. Par exemple, l'interface suivante, IExternalizable, fait partie du package flash.utils d'ActionScript 3.0. L'interface IExternalizable définit un protocole pour sérialiser un objet, ce qui correspond à la conversion d'un objet en un format qui convienne au stockage sur un périphérique ou au transport sur un réseau.

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

L'interface IExternalizable est déclarée avec le modificateur de contrôle d'accès `public`. Les définitions d'interfaces peuvent uniquement être modifiées à l'aide des spécificateurs de contrôle d'accès `public` et `internal`. Dans une définition d'interface, les déclarations de méthodes ne peuvent pas comporter de spécificateur de contrôle d'accès.

ActionScript 3.0 respecte la convention de nom selon laquelle les noms des interfaces débutent par un `I` majuscule, mais vous pouvez utiliser tout identificateur autorisé comme nom d'interface. Les définitions d'interfaces sont souvent placées au niveau supérieur d'un package. Les définitions d'interfaces ne peuvent pas être placées dans une définition de classe ou dans une autre définition d'interface.

Une interface peut étendre une ou plusieurs autres interfaces. Par exemple, l'interface IExample étend l'interface IExternalizable :

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

Toute classe qui implémente l'interface IExample doit comporter non seulement les implémentations de la méthode `extra()`, mais aussi celles des méthodes `writeExternal()` et `readExternal()` héritées de l'interface IExternalizable.

Implémentation d'une interface dans une classe

La classe est le seul élément du langage ActionScript 3.0 qui puisse implémenter une interface. Pour implémenter une ou plusieurs interfaces, on utilise le mot-clé `implements` dans une déclaration de classe. L'exemple suivant définit deux interfaces, IAlpha et IBeta, ainsi qu'une classe, Alpha, qui les implémente toutes deux :

```
interface IAlpha
{
    function foo(str:String):String;
}

interface IBeta
{
    function bar():void;
}

class Alpha implements IAlpha, IBeta
{
    public function foo(param:String):String {}
    public function bar():void {}
}
```

Dans une classe qui implémente une interface, les méthodes implémentées doivent :

- Utiliser l'identificateur de contrôle d'accès `public`.
- Utiliser le même nom que la méthode de l'interface.
- Avoir le même nombre de paramètres, chacun d'eux étant du type de données correspondant à celui du paramètre équivalent dans la méthode de l'interface.
- Utiliser le même type de retour.

```
public function foo(param:String):String {}
```

Vous disposez toutefois d'une certaines souplesse pour le nom des paramètres des méthodes que vous implémentez. Bien que le nombre et le type de données des paramètres de la méthode implémentée doivent correspondre à ceux de la méthode de l'interface, il n'est pas obligatoire que les noms des paramètres soient identiques. Par exemple, dans l'exemple ci-dessus, le paramètre de la méthode `Alpha.foo()` est appelé `param`.

Par contre, le paramètre correspondant est appelé `str` dans la méthode de l'interface `IAlpha.foo()` :

```
function foo(str:String):String;
```

Les valeurs par défaut des paramètres offrent également une certaine souplesse. La définition d'une interface peut comporter des déclarations de fonctions avec des valeurs par défaut pour les paramètres. Une méthode qui implémente l'une de ces déclarations de fonction doit disposer d'une valeur par défaut pour le ou les paramètres. Cette valeur doit être du même type de données que celle qui est spécifiée dans la définition de l'interface, mais ce n'est pas forcément le cas pour la valeur réelle. Par exemple, le code ci-dessous définit une interface contenant une méthode dont le paramètre a la valeur 3 par défaut :

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

La définition de classe suivante implémente l'interface Igamma, mais utilise une autre valeur par défaut pour le paramètre :

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

Cette souplesse est due au fait que les règles d'implémentation d'une interface sont spécifiquement conçues afin d'assurer une compatibilité des types de données ; il n'est pas nécessaire, pour ce faire, d'exiger des noms et des valeurs par défaut identiques pour les paramètres.

Héritage

L'héritage est une forme de réutilisation du code qui permet aux programmeurs de développer de nouvelles classes à partir de classes existantes. Les classes existantes sont alors fréquemment appelées *classes de base* ou *superclasses*, alors que les nouvelles classes sont généralement appelées *sous-classes*. L'un des principaux avantages de l'héritage est qu'il permet de réutiliser le code d'une classe de base sans modifier le code existant. De plus, l'héritage ne nécessite pas de modifier les modes d'interaction des autres classes avec la classe de base. Plutôt que de modifier une classe existante, qui est peut-être soigneusement testée et déjà utilisée, l'héritage permet de traiter cette classe comme un module intégré qui peut être étendu à l'aide de propriétés et de méthodes supplémentaires. C'est pourquoi on utilise le mot-clé `extends` pour indiquer qu'une classe hérite d'une autre classe.

L'héritage permet également de tirer parti du *polymorphisme* du code. Le polymorphisme est la possibilité d'utiliser un nom de méthode unique pour une méthode qui se comporte différemment en fonction des types de données qu'elle reçoit. Par exemple, supposons une classe de base appelée Shape, avec deux sous-classes appelées Circle et Square. La classe Shape définit une méthode appelée `area()` qui renvoie la surface de Shape. Si vous avez implémenté le polymorphisme, vous pouvez appeler la méthode `area()` pour les objets de type Circle ou Square et lui faire exécuter le calcul correct. L'héritage autorise le polymorphisme en permettant aux sous-classes d'hériter et de définir, ou *override*, les méthodes de la classe de base. Dans l'exemple suivant, la méthode `area()` est redéfinie par les classes Circle et Square :

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

Dans la mesure où chaque classe définit un type de données, l'utilisation de l'héritage crée un rapport spécial entre une classe de base et une classe qui l'étend. Une sous-classe possède obligatoirement toutes les propriétés de sa classe de base, ce qui signifie qu'il est toujours possible de substituer une occurrence d'une sous-classe à une occurrence de la classe de base. Par exemple, si une méthode définit un paramètre du type Shape, il est parfaitement valable de lui transmettre un argument du type Circle, car Circle étend Shape, comme on peut le voir ci-dessous :

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

Propriétés et héritage des occurrences

Qu'elle soit définie à l'aide du mot-clé `function`, `var` ou `const`, une propriété d'occurrence est héritée par toutes les sous-classes tant que cette propriété n'est pas déclarée avec l'attribut `private` dans la classe de base. Par exemple, la classe Event d'ActionScript 3.0 possède des sous-classes nombreuses qui héritent de propriétés communes à tous les objets événements.

Pour certains types d'événements, la classe Event contient toutes les propriétés nécessaires pour définir l'événement. Ces types d'événements ne nécessitent pas de propriétés d'occurrence au-delà de celles qui sont définies dans la classe Event. L'événement `complete`, qui est déclenché lorsque des données ont été chargées avec succès, et l'événement `connect`, qui se produit lorsqu'une connexion réseau a été établie, sont des exemples de ce type d'événement.

L'exemple suivant est extrait de la classe Event. Il montre certaines propriétés et méthodes dont les sous-classes héritent. Comme elles sont héritées, ces propriétés sont accessibles par une occurrence de n'importe quelle sous-classe.

```
public class Event
{
    public function get type():String;
    public function get bubbles():Boolean;
    ...

    public function stopPropagation():void {}
    public function stopImmediatePropagation():void {}
    public function preventDefault():void {}
    public function isDefaultPrevented():Boolean {}
    ...
}
```

D'autres types d'événements nécessitent des propriétés uniques qui ne sont pas disponibles dans la classe Event. Ces événements sont définis à l'aide de sous-classes de la classe Event, ce qui permet d'ajouter de nouvelles propriétés à celles de cette classe Event. La classe MouseEvent est un exemple de sous-classe de ce type. Elle ajoute des propriétés uniques aux événements associés à un mouvement ou à un clic de souris, tels que les événements `mouseMove` et `click`. L'exemple suivant est extrait de la classe MouseEvent. Il montre la définition des propriétés inhérentes à la sous-classe parce qu'absentes de la classe de base.

```
public class MouseEvent extends Event
{
    public static const CLICK:String= "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

Spécificateurs de contrôle d'accès et héritage

Si une propriété est déclarée avec le mot-clé `public`, elle est visible de n'importe quel point du code. Cela signifie que le mot-clé `public` n'introduit aucune restriction sur l'héritage des propriétés, contrairement aux mots-clés `private`, `protected` et `internal`.

Si une propriété est déclarée avec le mot-clé `private`, elle n'est visible qu'à partir de la classe qui la définit. Autrement dit les sous-classes n'en héritent pas. Ce comportement est différent de celui des versions antérieures d'ActionScript où le mot-clé `private` se comportait plutôt comme le mot-clé `protected` d'ActionScript 3.0.

Le mot-clé `protected` indique qu'une propriété est visible non seulement à partir de la classe qui la définit, mais aussi à partir de toutes les sous-classes de celle-ci. Contrairement au mot-clé `protected` en Java, en ActionScript 3.0 le mot-clé`protected` ne rend pas une propriété visible à partir de toutes les autres classes du même package. En ActionScript 3.0, seules les sous-classes peuvent accéder à une propriété déclarée avec le mot-clé `protected`. De plus, une propriété protégée est visible à partir d'une sous-classe même si celle-ci ne se trouve pas dans le même package que sa classe de base.

Pour limiter la visibilité d'une propriété au package dans lequel elle est définie, vous pouvez soit utiliser le mot-clé `internal`, soit n'utiliser aucun spécificateur de contrôle d'accès. Le spécificateur de contrôle d'accès `internal` est appliqué par défaut si vous n'en indiquez aucun. Seule une sous-classe résidant dans le même package pourra hériter d'une propriété marquée comme internal. Seule une sous-classe résidant dans le même package peut hériter d'une propriété désignée comme `internal`.

L'exemple suivant montre comment chaque spécificateur de contrôle d'accès affecte l'héritage dans et au-delà des package. Le code ci-dessous définit une classe principale d'application appelée AccessControl et deux autres classes, Base et Extender. La classe Base se trouve dans un package appelé foo et la classe Extender, qui est une sous-classe de la classe Base, dans un package appelé bar. La classe AccessControl n'importe que la classe Extender et crée une occurrence de celle-ci qui tente d'accéder à une variable appelée `str`, définie dans la classe Base. La variable `str` est déclarée comme `public`, si bien que le code est compilé et exécuté comme l'illustre l'exemple ci-dessous :

```
// Base.as in a folder named foo
package foo
{
    public class Base
    {
        public var str:String = "hello"; // change public on this line
    }
}

// Extender.as in a folder named bar
package bar
{
    import foo.Base;
    public class Extender extends Base
    {
        public function getString():String {
            return str;
        }
    }
}

// main application class in file named AccessControl.as
package
{
    import flash.display.MovieClip;
    import bar.Extender;
    public class AccessControl extends MovieClip
    {
        public function AccessControl()
        {
            var myExt:Extender = new Extender();
            trace(myExt.str);// error if str is not public
            trace(myExt.getString()); // error if str is private or internal
        }
    }
}
```

Pour voir l'effet des autres spécificateurs de contrôle d'accès lors de la compilation et de l'exécution de cet exemple, changez le spécificateur de contrôle de la variable `str` à `private`, `protected` ou `internal` après avoir supprimé ou mis en commentaire la ligne suivante dans la classe `AccessControl` :

```
trace(myExt.str);// error if str is not public
```

Redéfinition des variables impossible

Les propriétés déclarées à l'aide des mots-clés `var` ou `const` sont héritées mais ne peuvent pas être redéfinies. Redéfinir, ou forcer, une propriété au sens de « override » signifie redéfinir cette propriété dans une sous-classe. Les accesseurs get et set (c'est-à-dire les propriétés déclarées avec le mot-clé `function`) constituent le seul type de propriété qu'il est possible de redéfinir. Bien qu'il soit impossible de redéfinir une variable d'occurrence, vous pouvez obtenir le même résultat par la création des méthodes de lecture et de définition pour cette variable d'occurrence et en forçant ces méthodes.

Redéfinition des méthodes

Redéfinir une méthode signifie redéfinir le comportement d'une méthode héritée. Les méthodes statiques ne sont pas héritées et ne peuvent donc pas être redéfinies. Toutefois, les sous-classes héritent des méthodes d'occurrence et il est donc possible de redéfinir celles-ci sous réserve de deux conditions :

- La méthode d'occurrence ne doit pas être déclarée avec le mot-clé `final` dans la classe de base. Lorsqu'il est utilisé avec une méthode d'occurrence, le mot-clé `final` indique que le programmeur veut empêcher les sous-classes de redéfinir cette méthode.
- La méthode d'occurrence ne doit pas être déclarée avec le contrôle d'accès `private` dans la classe de base. Si une méthode est désignée comme `private` dans la classe de base, il est inutile d'utiliser le mot-clé `override` lors de la définition d'une méthode portant le même nom dans la sous-classe, puisque la méthode de la classe de base n'est pas visible à partir de la sous-classe.

  Pour redéfinir une méthode d'occurrence correspondant à ces critères, la définition de la méthode dans la sous-classe doit utiliser le mot-clé `override` et correspondre, comme défini ci-dessous, à la version de cette méthode dans la superclasse :

- La méthode de redéfinition doit avoir le même niveau de contrôle d'accès que celle de la classe de base. Les méthodes définies comme internes doivent avoir le même niveau de contrôle d'accès que celles qui n'ont pas de spécificateur de contrôle d'accès.
- La méthode de redéfinition doit avoir le même nombre de paramètres que celle de la classe de base.
- Les paramètres de la méthode de redéfinition doivent avoir les mêmes annotations de type de données que ceux de la méthode de la classe de base.
- La méthode de redéfinition doit avoir le même type de renvoi que celle de la classe de base.

Toutefois, il n'est pas nécessaire que les noms des paramètres de la méthode de redéfinition correspondent à ceux des paramètres de la classe de base, tant qu'il y a une correspondance entre le nombre de paramètres et leurs types de données.

Instruction super

Il arrive fréquemment que les programmeurs souhaitent compléter le comportement de la méthode de superclasse qu'ils redéfinissent, plutôt que de remplacer ce comportement. Il est donc nécessaire de disposer d'un mécanisme permettant à une méthode d'une sous-classe d'appeler sa propre superclasse. Ce mécanisme est assuré par l'instruction `super` qui contient une référence à la superclasse immédiate. L'exemple suivant définit une classe appelée Base, qui contient la méthode appelée `thanks()` et une sous-classe de Base appelée Extender, qui redéfinit la méthode `thanks()`. La méthode `Extender.thanks()` utilise l'instruction `super` pour appeler `Base.thanks()`.

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

Redéfinition des accesseurs Get et Set

Bien qu'il soit impossible de redéfinir les variables définies dans une superclasse, il est possible de redéfinir les accesseurs Get et Set. Par exemple, le code suivant redéfinit un accesseur Get appelé `currentLabel` qui est défini dans la classe MovieClip d'ActionScript 3.0 :

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

Le résultat de l'instruction `trace()` dans le constructeur de la classe OverrideExample est `Override: null`, ce qui montre que cet exemple a réussi à redéfinir la propriété héritée`currentLabel`.

Propriétés statiques non héritées

Les sous-classes n'héritent pas des propriétés statiques. Ces dernières ne sont donc pas accessibles par une occurrence d'une sous-classe. Une propriété statique n'est accessible que par le biais de l'objet classe dans lequel elle est définie. Par exemple, le code suivant définit une classe de base appelée Base et une sous-classe appelée Extender, qui étend la classe Base. Une variable statique appelée `test` est définie dans la classe Base. Le code de l'extrait ci-dessous ne peut être compilé en mode strict et génère une erreur d'exécution en mode standard.

```
package {
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// error
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base { }
```

La seule façon d'accéder à la variable statique `test` est par l'objet classe, comme le montre le code suivant :

```
Base.test;
```

Il est toutefois permis de définir une propriété d'occurrence ayant le même nom qu'une propriété statique. Cette propriété d'occurrence peut être définie dans la même classe que la propriété statique, ou dans une sous-classe. Par exemple, la classe Base de l'exemple précédent peut comporter une propriété d'occurrence appelée `test`. Le code suivant peut être compilé et exécuté normalement, car la classe Extender hérite de la propriété d'occurrence. Ce code peut aussi être compilé et exécuté normalement si la définition de la variable d'occurrence test est déplacée (mais non copiée) dans la classe Extender.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

Propriétés statiques et chaîne de portée

Bien que les propriétés statiques ne soient pas héritées, elles figurent dans la chaîne de portée de la classe qui les définit ainsi que de toute sous-classe de celle-ci. C'est pourquoi on dit que les propriétés statiques sont *dans la portée* à la fois de la classe dans laquelle elles sont définies et des sous-classes de celle-ci. Cela signifie qu'une propriété statique est directement accessible à partir du corps de la classe qui la définit et de toute sous-classe de celle-ci.

L'exemple suivant modifie les classes définies dans l'exemple précédent pour montrer que la variable statique `test`, définie dans la classe Base, est dans la portée de la classe Extender. Autrement dit, la classe Extender peut accéder à la variable statique `test` sans qu'il ne soit nécessaire de faire précéder le nom de celle-ci du nom de la classe qui définit `test`.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base
{
    public function Extender()
    {
        trace(test); // output: static
    }

}
```

Si une propriété d'occurrence est définie avec le même nom qu'une propriété statique de la même classe ou d'une superclasse, la propriété d'occurrence est prioritaire dans la chaîne de portée. On dit alors que la propriété d'occurrence *fait de l'ombre* à la propriété statique, ce qui signifie que la valeur de la propriété d'occurrence est utilisée à la place de celle de la propriété statique. Par exemple, le code ci-dessous montre que si la classe Extender définit une variable d'occurrence appelée `test`, l'instruction `trace()` utilise la valeur de la variable d'occurrence à la place de celle de la variable statique.

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base
{
    public static var test:String = "static";
}

class Extender extends Base
{
    public var test:String = "instance";
    public function Extender()
    {
        trace(test); // output: instance
    }

}
```

Rubriques avancées

Historique de la prise en charge de la programmation orientée objets en ActionScript

ActionScript 3.0 est une évolution des versions antérieures d'ActionScript. C'est pourquoi il peut être utile de comprendre l'évolution du modèle d'objet en ActionScript. ActionScript était à l'origine un simple mécanisme de script pour les premières versions de Flash Professional. Les programmeurs ont alors commencé à développer des applications de plus en plus complexes avec ActionScript. En réponse aux besoins de ces programmeurs, le langage de chaque nouvelle version a connu des ajouts destinés à faciliter la création d'applications complexes.

ActionScript 1.0

ActionScript 1.0 est le langage utilisé dans les versions 6 et antérieures de Flash Player. Même à ce stade précoce de développement, le modèle d'objet d'ActionScript était basé sur le concept de l'objet comme type fondamental de données. Un objet ActionScript est un type de données composite doté d'un groupe de *propriétés*. Dans le contexte d'un modèle d'objet, le terme *propriétés* désigne tout ce qui est affecté à un objet (variables, fonctions ou méthodes).

Bien que cette première génération d'ActionScript ne prenne pas en charge la définition de classes avec le mot-clé `class`, elle permet de définir une classe à l'aide d'un type spécial d'objet appelé objet prototype. Au lieu d'utiliser un mot-clé `class` pour créer une définition de classe abstraite qui sera ensuite instanciée en objets concrets (à l'instar des langages reposant sur des classes, comme Java ou C++), les langages basés sur le prototypage, tel ActionScript 1.0, utilisent un objet existant comme modèle (ou prototype) d'autres objets. Alors que les objets des langages basés sur la notion de classes peuvent pointer sur une classe qui leur sert de modèle, les objets des langages basés sur la notion de prototypes pointent sur un autre objet, leur prototype, qui leur sert de modèle.

Pour créer une classe en ActionScript 1.0, il est nécessaire de définir une fonction constructeur pour cette classe. En ActionScript, les fonctions sont des objets réels et non pas de simples définitions abstraites. La fonction constructeur sert alors d'objet prototype pour les occurrences de cette classe. Le code suivant crée une classe appelée Shape et définit une propriété appelée `visible` qui a la valeur `true` par défaut.

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

Cette fonction constructeur définit une classe Shape qui va être instanciée à l'aide de l'opérateur `new`, comme suit :

```
myShape = new Shape();
```

De même que l'objet de la fonction constructeur `Shape()` sert de prototype pour les occurrences de la classe Shape, il peut également servir pour les sous-classes de Shape, c'est-à-dire d'autres classes qui prolongent la classe Shape.

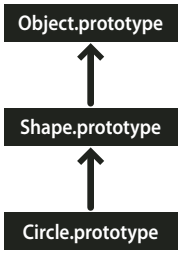
La création d'une classe qui est une sous-classe de la classe Shape est un processus en deux étapes. D'abord, créer la classe en définissant une fonction constructeur pour cette classe, comme suit :

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

Ensuite, utiliser l'opérateur `new` pour déclarer que la classe Shape est le prototype de la classe Circle. Par défaut, toute nouvelle classe créée utilise la classe Object comme prototype, si bien que `Circle.prototype` contient alors un objet générique (une occurrence de la classe Object). Pour spécifier que le prototype de Circle est Shape et non pas Object, utilisez le code suivant pour changer la valeur de `Circle.prototype` afin qu'elle contienne un objet Shape et non plus un objet générique :

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

La classe Shape et la classe Circle sont maintenant liées par une relation d'héritage communément appelée *chaînage du prototype*. Le schéma illustre la relation au sein d'un chaînage de prototype :



La classe de base, à la fin de chaque chaînage de prototype, est la classe Object. La classe Object contient une propriété statique appelée `Object.prototype` qui pointe sur l'objet prototype de base pour tous les objets créés en ActionScript 1.0. Dans l'exemple de chaînage de prototype, l'objet suivant est l'objet Shape. En effet, la propriété `Shape.prototype` n'a jamais été explicitement définie, si bien qu'elle contient encore un objet générique (une occurrence de la classe Object). Le lien final de ce chaînage est la classe Circle qui est liée à son prototype, la classe Shape (la propriété `Circle.prototype` contient un objet Shape).

Si vous créez une occurrence de la classe Circle, comme dans l'exemple ci-dessous, l'occurrence hérite du chaînage de prototype de la classe Circle :

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

Pour rappel, l'exemple contient une propriété appelée `visible`, qui est membre de la classe Shape. La propriété `visible` n'existe pas comme partie de l'objet`myCircle`, mais uniquement comme membre de l'objet Shape ; et pourtant la ligne de code suivante produit la valeur `true` :

```
trace(myCircle.visible); // output: true
```

En remontant le chaînage du prototype, le moteur d'exécution est en mesure de vérifier que l'objet `myCircle` hérite de la propriété `visible`. Lorsque ce code est exécuté, le moteur d'exécution recherche d'abord dans les propriétés de l'objet `myCircle` une propriété appelée `visible`, mais ne la trouve pas. Il examine alors l'objet `Circle.prototype`, mais ne trouve toujours pas la propriété `visible`. En continuant à remonter le chaînage du prototype, il finit par trouver la propriété `visible` définie dans l'objet `Shape.prototype` et affiche sa valeur.

Par souci de simplicité, la présente section omet un grand nombre de détails et de subtilités du chaînage de prototype, puisqu'il s'agit simplement de vous aider à comprendre le modèle d'objet en ActionScript 3.0.

ActionScript 2.0

Avec ActionScript 2.0 ont été introduits de nouveaux mots-clés tels que `class`, `extends`, `public` et `private` qui permettaient de définir des classes selon une méthode bien connue de toute personne ayant travaillé avec les langages basés sur des classes, comme Java et C++. Il est important de comprendre que le mécanisme sous-jacent d'héritage n'a pas changé entre ActionScript 1.0 et ActionScript 2.0. Le passage à ActionScript 2.0 a consisté simplement à ajouter une nouvelle syntaxe pour la définition des classes. Le chaînage du prototype fonctionne de la même façon dans ces deux versions du langage.

La nouvelle syntaxe introduite par ActionScript 2.0 est représentée dans l'exemple ci-dessous. Elle permet de définir des classes d'une façon que la plupart des programmeurs considèrent comme plus intuitive :

```
// base class
class Shape
{
var visible:Boolean = true;
}
```

Vous pouvez remarquer qu'ActionScript 2.0 a également introduit les annotations de type destinées à une vérification des types à la compilation. Elles permettent de déclarer que la propriété `visible` de l'exemple précédent ne doit contenir qu'une valeur booléenne. Le nouveau mot-clé `extends` simplifie lui aussi le processus de création d'une sous-classe. Dans l'exemple suivant, ce qui nécessitait deux étapes en ActionScript 1.0 est accompli en une seule, à l'aide du mot-clé `extends` :

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

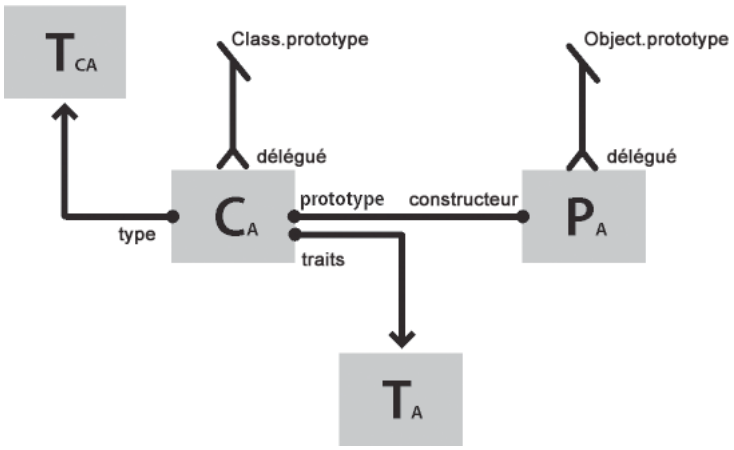
Le constructeur est maintenant déclaré dans le cadre de la définition de classe, et les propriétés `id` et `radius` de la classe doivent elles aussi être déclarées explicitement.

ActionScript 2.0 a également ajouté la prise en charge de la définition des interfaces, qui permet de rendre plus perfectionnés des programmes orientés objet, à l'aide de protocoles formellement définis pour la communication entre les objets.

Objet de classe en ActionScript 3.0

Un paradigme courant en programmation orientée objets, en particulier avec Java et C++, fait appel à des classes pour définir des types d'objets. Les langages de programmation qui adoptent ce paradigme ont aussi tendance à utiliser des classes pour construire des occurrences du type de données défini par la classe. ActionScript utilise des classes pour ces deux raisons, mais ses origines de langage basé sur des prototypes lui confèrent une caractéristique intéressante. Pour chaque définition de classe, ActionScript crée un objet de classe spécial qui autorise le partage du comportement et de l'état. Toutefois, pour de nombreux programmeurs en ActionScript, cette distinction n'aura aucune implication pratique sur le codage. En effet, ActionScript 3.0 est conçu de telle sorte qu'il est possible de créer des applications perfectionnées orientées objet sans utiliser, et sans même comprendre, ces objets de classe spéciaux.

Le schéma suivant montre la structure d'un objet de classe représentant une classe simple appelée A, définie par l'instruction `class A`.



Chaque rectangle du schéma représente un objet. Chaque objet du schéma est marqué d'un caractère en indice pour signaler qu'il appartient à la classe A. L'objet de classe (CA) contient des références à un certain nombre d'autres objets importants. L'objet traits des occurrences (TA) enregistre les propriétés des occurrences qui sont définies dans une définition de classe. Un objet traits de la classe (TCA) représente le type interne de la classe et enregistre les propriétés statiques définies par la classe (le caractère C en indice signifie « classe »). L'objet prototype (PA) désigne toujours l'objet de classe auquel il était associé à l'origine par la propriété `constructor`.

Objet traits

L'objet traits est une nouveauté d'ActionScript 3.0, implémentée pour des raisons de performances. Dans les versions précédentes d'ActionScript, la recherche d'un nom pouvait demander beaucoup de temps pendant que Flash Player remontait le chaînage du prototype. En ActionScript 3.0, la recherche d'un nom est beaucoup plus rapide et efficace, car les propriétés héritées sont copiées des superclasses dans l'objet traits des sous-classes.

L'objet traits n'est pas directement accessible par code, mais sa présence se manifeste par l'amélioration des performances et de l'utilisation de la mémoire. L'objet traits fournit à la machine virtuelle AVM2 des informations détaillées sur la disposition et le contenu d'une classe. Ces informations permettent à AVM2 de réduire nettement le temps d'exécution, car elle peut fréquemment générer des instructions machine directes pour accéder à des propriétés ou appeler des méthodes directement, sans effectuer au préalable une longue recherche de nom.

Grâce à l'objet traits, l'espace occupé en mémoire par un objet peut être nettement moins importante qu'avec les versions antérieures d'ActionScript. Par exemple, si une classe est scellée (c'est-à-dire si elle n'est pas déclarée comme dynamique), une occurrence de la classe ne nécessite pas de recherche par calcul d'adresse pour les propriétés ajoutées dynamiquement ; il lui suffit d'un pointeur sur l'objet traits et de quelques emplacements pour les propriétés fixes définies dans la classe. En conséquence, pour un objet qui nécessitait 100 octets en mémoire avec ActionScript 2.0, 20 suffiront avec ActionScript 3.0.

Remarque : *l'objet traits est un élément d'implémentation interne et il n'est pas garanti qu'il ne changera pas, ou même qu'il ne disparaîtra pas, dans les futures versions d'ActionScript.*

Objet prototype

En ActionScript, chaque objet de classe possède une propriété appelée `prototype`, qui est une référence à l'objet prototype de cette classe. L'objet prototype est un héritage des premières versions d'ActionScript, basées sur des prototypes. Pour plus d'informations, voir Historique de la prise en charge de la programmation orientée objets dans ActionScript.

La propriété `prototype` est en lecture seule et ne peut donc pas être modifiée pour pointer sur d'autres objets. A l'inverse, dans les anciennes versions d'ActionScript, la propriété `prototype` pouvait être réaffectée pour pointer sur une autre classe. Bien que la propriété `prototype` soit en lecture seule, ce n'est pas le cas de l'objet prototype à laquelle elle fait référence. En d'autres termes, de nouvelles propriétés peuvent être ajoutées à l'objet prototype. Les propriétés ajoutées à l'objet prototype sont partagées avec toutes les autres occurrences de la classe.

Le chaînage de prototype, qui était le seul mécanisme d'héritage des versions antérieures d'ActionScript, n'a plus qu'un rôle secondaire en ActionScript 3.0. Le principal mécanisme d'héritage, l'héritage des propriétés fixes, est géré de façon interne par l'objet traits. Une propriété fixe est une variable ou une méthode définie dans le cadre d'une définition de classe. L'héritage des propriétés fixes est également appelé héritage des classes, car c'est le mécanisme d'héritage qui est associé aux mots-clés `class`, `extends` et `override`.

Le chaînage de prototype représente un autre mécanisme d'héritage qui est plus dynamique que l'héritage des propriétés fixes. Il est possible d'ajouter des propriétés à l'objet prototype d'une classe, non seulement dans le cadre de la définition de classe, mais aussi lors de l'exécution, par la propriété `prototype` de l'objet de classe. Notez toutefois que si le compilateur est en mode strict, il peut être impossible d'accéder aux propriétés ajoutées à un objet prototype, sauf si vous déclarez une classe avec le mot-clé `dynamic`.

La classe Object est un bon exemple d'une classe dont plusieurs propriétés sont attachées à l'objet prototype. Les méthodes `toString()` et `valueOf()` de la classe Object sont en fait des fonctions affectées à des propriétés de l'objet prototype de la classe Object. Voici un exemple de l'aspect théorique de la déclaration de ces méthodes (l'implémentation réelle est légèrement différente en raison des détails pratiques d'implémentation) :

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

Comme nous l'avons déjà mentionné, il est possible d'attacher une propriété à l'objet prototype d'une classe en dehors de la définition de cette classe. Par exemple, la méthode`toString()` peut également être définie hors de la définition de la classe Object, comme suit :

```
Object.prototype.toString = function()
{
    // statements
};
```

Toutefois, contrairement à l'héritage des propriétés fixes, l'héritage du prototype ne nécessite pas le mot-clé `override` pour redéfinir une méthode de sous-classe. Par exemple, pour redéfinir la méthode `valueOf()` dans une sous-classe de la classe Object, trois options sont possibles. Premièrement, vous pouvez définir une méthode `valueOf()` dans l'objet prototype de la sous-classe, à l'intérieur de la définition de classe. Le code suivant crée une sous-classe d'Object appelée Foo et redéfinit la méthode `valueOf()` dans la définition de classe de l'objet prototype de Foo. Chaque classe héritant de la classe Object, il n'est pas nécessaire d'utiliser le mot-clé `extends`.

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

Deuxièmement, vous pouvez définir une méthode `valueOf()` dans l'objet prototype de Foo en-dehors de la définition de classe, comme le montre le code ci-dessous :

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

Troisièmement, vous pouvez définir une propriété fixe appelée `valueOf()` dans la classe Foo. Cette technique diffère des précédentes dans la mesure où elle mêle l'héritage des propriétés fixes à l'héritage du prototype. Pour redéfinir `valueOf()` à partir d'une sous-classe de Foo, il est nécessaire d'utiliser le mot-clé `override`. Le code ci-dessous montre `valueOf()` définie comme propriété fixe dans Foo :

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

Espace de noms d'ActionScript 3

L'existence de deux mécanismes d'héritage séparés, l'héritage des propriétés fixes et l'héritage du prototype, provoque un problème intéressant de compatibilité par rapport aux propriétés et méthodes des classe de base. La compatibilité avec la spécification du langage ECMAScript sur laquelle est basé ActionScript nécessite l'utilisation de l'héritage de prototype, si bien que les propriétés et les méthodes d'une classe de base sont définies dans l'objet prototype de cette classe. Mais par ailleurs, la compatibilité avec ActionScript 3.0 nécessite l'utilisation de l'héritage des propriétés fixes, si bien que les propriétés et méthodes d'une classe de base sont spécifiées dans la définition de classe, à l'aide des mots-clés `const`, `var` et `function`. De plus, l'utilisation des propriétés fixes plutôt que les versions du prototype est susceptible de permettre une nette amélioration des performances à l'exécution.

Pour résoudre ce problème, ActionScript 3.0 utilise les deux mécanismes d'héritage (propriétés fixes et prototype) pour les classes de base. Chaque classe de base contient deux jeux de propriétés et de méthodes. Un jeu est défini dans l'objet prototype pour assurer la compatibilité avec les spécifications d'ECMAScript et l'autre est défini avec les propriétés fixes et l'espace de noms d'AS3,0, pour assurer la compatibilité avec ActionScript 3.0.

L'espace de noms d'AS3 représente un mécanisme fort pratique pour choisir entre les deux jeux de propriétés et de méthodes. Si vous n'utilisez pas l'espace de noms d'AS3, une occurrence d'une classe de base hérite des propriétés et des méthodes définies dans l'objet prototype de cette classe de base. Si par contre vous utilisez l'espace de noms d'AS3, une occurrence d'une classe de base hérite des versions d'AS3, car les propriétés fixes sont toujours préférées aux propriétés du prototype. En d'autres termes, lorsqu'une propriété fixe est disponible, elle est toujours utilisée à la place d'une propriété du prototype ayant le même nom.

Il est possible d'utiliser sélectivement la version de l'espace de noms d'AS3 d'une propriété ou d'une méthode, en la qualifiant à l'aide de l'espace de noms d'AS3. Par exemple, le code ci-dessous utilise la version AS3 de la méthode `Array.pop()` :

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

Vous pouvez aussi faire appel à la directive `use namespace` afin d'ouvrir l'espace de noms d'AS3 pour toutes les définitions figurant dans un bloc de code. Par exemple, le code ci-dessous utilise la directive `use namespace` afin d'ouvrir l'espace de noms d'AS3 pour les méthodes `pop()` et `push()`.

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 comporte aussi des options de compilation pour chaque jeu de propriétés, ce qui permet d'appliquer l'espace de noms d'AS3 au programme entier. L'option de compilation `-as3` représente l'espace de noms d'AS3 et l'option de compilation `-es` représente l'option d'héritage du prototype (`es` correspond à ECMAScript). Pour ouvrir l'espace de noms d'AS3 pour tout le programme, définissez l'option de compilation `-as3` sur `true` et l'option de compilation `-es` sur `false`. Pour utiliser les versions du prototype, définissez les options de compilation sur les valeurs opposées. Les options de compilation par défaut de Flash Builder et Flash Professional sont `-as3 = true` et `-es = false`.

Si vous prévoyez d'étendre l'une des classes de base et de redéfinir des méthodes, vous devez comprendre comment l'espace de noms d'AS3 affecte la façon de déclarer une méthode redéfinie. Si vous utilisez l'espace de noms d'AS3, toute redéfinition d'une méthode d'une classe de base doit également utiliser l'espace de noms d'AS3, ainsi que l'attribut `override`. Si vous n'utilisez pas l'espace de noms d'AS3 et voulez redéfinir une méthode d'une classe de base dans une sous-classe, vous ne devez utiliser ni l'espace de noms d'AS3 dans cette redéfinition, ni l'attribut `override`.

Exemple : GeometricShapes

L'exemple d'application GeometricShapes montre comment il est possible d'appliquer un certain nombre de concepts et fonctionnalités orientés objet à l'aide d'ActionScript 3.0 et, en particulier :

- Définition des classes
- Extension des classes
- Polymorphisme et le mot-clé `override`
- Définition, extension et implémentation des interfaces

Cet exemple comprend aussi une « méthode usine (Factory) » qui crée des occurrences de classes, montrant ainsi comment déclarer une valeur de retour comme occurrence d'une interface et utiliser l'objet ainsi renvoyé de manière générique.

Pour obtenir les fichiers de cet exemple d'application, voir la page www.adobe.com/go/learn_programmingAS3samples_flash_fr. Les fichiers de l'application GeometricShapes se trouvent dans le dossier Samples/GeometricShapes. L'application se compose des fichiers suivants :

Fichier	Description
GeometricShapes.mxml ou GeometricShapes.fla	Le fichier d'application principal dans Flash (FLA) ou Flex (MXML).
com/example/programmingas3/geometricshapes/IGeometricShape.as	Interface de base définissant les méthodes qui doivent être implémentées par toutes les classes de l'application GeometricShapes.
com/example/programmingas3/geometricshapes/IPolygon.as	Interface définissant les méthodes qui doivent être implémentées par les classes de l'application GeometricShapes qui comportent plusieurs côtés.
com/example/programmingas3/geometricshapes/RegularPolygon.as	Type de forme géométrique dont les côtés sont de longueur égale et positionnés symétriquement autour du centre de la forme.
com/example/programmingas3/geometricshapes/Circle.as	Type de forme géométrique qui définit un cercle.
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	Sous-classe de RegularPolygon qui définit un triangle équilatéral.
com/example/programmingas3/geometricshapes/Square.as	Sous-classe de RegularPolygon qui définit un carré.
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	Classe contenant une méthode usine (Factory) pour créer des formes à partir d'un type et d'une taille de forme.

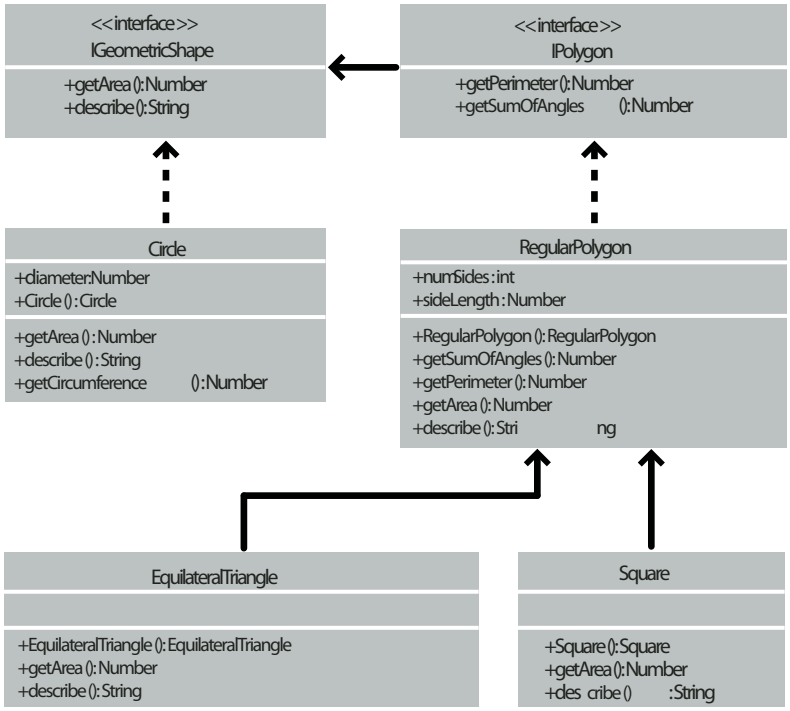
Définition des classes de GeometricShapes

L'application GeometricShapes permet de spécifier un type de forme géométrique et la taille de cette dernière. Elle renvoie alors la description de la forme, sa surface et son périmètre.

L'interface utilisateur de l'application est banale : elle se compose de quelques contrôles permettant de sélectionner le type de forme, de définir la taille et d'afficher la description. De fait, la partie intéressante de cette application est la face cachée de l'iceberg : la structure des classes et des interfaces elles-mêmes.

Cette application traite des formes géométriques, mais elle ne les affiche pas graphiquement.

Les classes et interfaces qui définissent les formes géométriques de cet exemple sont représentées dans le schéma ci-dessous, en notation UML (Unified Modeling Language) :



GeometricShapes Example Classes

Définition d'un comportement commun à l'aide d'interfaces

Cette application GeometricShapes gère trois types de formes : cercles, carrés et triangles équilatéraux. La structure des classes de GeometricShapes commence par une interface très simple, IGeometricShape, qui répertorie des méthodes communes aux trois types de formes :

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

L'interface définit deux méthodes : la méthode `getArea()` qui calcule et renvoie la surface de la forme, et la méthode `describe()` qui assemble une description textuelle des propriétés de la forme.

Il est aussi souhaitable de connaître le périmètre de chaque forme. Toutefois, le périmètre d'un cercle est appelé circonférence et il est calculé de façon particulière, si bien que le comportement diverge de ceux d'un triangle ou d'un carré. Il existe cependant assez de similitudes entre les triangles, les carrés et les autres polygones pour qu'il soit logique de définir une nouvelle classe d'interface pour eux : IPolygon. L'interface IPolygon est également plutôt simple, comme on peut le constater :

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

Cette interface définit deux méthodes communes à tous les polygones : la méthode `getPerimeter()` qui mesure la longueur combinée de tous les côtés et la méthode `getSumOfAngles()` qui additionne tous les angles intérieurs.

L'interface IPolygon étend l'interface IGeometricShape, si bien que toute classe qui implémente l'interface IPolygon doit déclarer les quatre méthodes : les deux de l'interface IGeometricShape et les deux de l'interface IPolygon.

Définition des classes de formes

Dès que vous avez une vue claire des méthodes communes à chaque type de forme, vous pouvez définir les classes de formes elles-mêmes. En ce qui concerne le nombre de méthodes à implémenter, la forme la plus simple est la classe Circle :

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius * radius.
            var radius:Number = diameter / 2;
            return Math.PI * radius * radius;
        }

        public function getCircumference():Number
        {
            // The formula is Pi * diameter.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

La classe Circle implémente l'interface IGeometricShape. Elle doit donc comporter du code pour les méthodes `getArea()` et `describe()`. De plus, elle définit la méthode `getCircumference()` qui est unique à la classe Circle. La classe Circle déclare aussi une propriété, `diameter`, qui n'apparaîtra pas dans les autres classes, dédiées aux polygones.

Les deux autres types de formes, les carrés et les triangles équilatéraux, ont d'autres points communs : ils ont tous deux des côtés de longueur égale et il existe des formules communes pour calculer leurs périmètre et la somme de leurs angles intérieurs. En fait, ces formules communes s'appliquent à tout autre polygone régulier que vous définissez par la suite.

La classe RegularPolygon est la superclasse de la classe Square et de la classe EquilateralTriangle. Une superclasse permet de définir en un seul point des méthodes communes. Il n'est donc pas nécessaire de les définir séparément dans chaque sous-classe. Voici le code de la classe RegularPolygon :

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
```

```
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + "
degrees.\n";
            return desc;
        }
    }
}
```

La classe RegularPolygon déclare d'abord deux propriétés qui sont communes à tous les polygones réguliers : la longueur de chaque côté (la propriété `sideLength`) et le nombre de faces (la propriété `numSides`).

La classe RegularPolygon implémente l'interface IPolygon et déclare les quatre méthodes de l'interface IPolygon. Elle implémente deux d'entre elles, les méthodes `getPerimeter()` et `getSumOfAngles()` à l'aide de formules communes.

La formule de la méthode `getArea()` étant différente d'une forme à l'autre, la version de la méthode dans la classe de base ne peut pas comporter une logique commune dont hériteraient les méthodes des sous-classes. Elle renvoie donc simplement une valeur 0 par défaut pour indiquer que la surface n'a pas été calculée. Pour calculer correctement la surface de chaque forme, les sous-classes de la classe RegularPolygon doivent redéfinir elles-mêmes la méthode `getArea()`.

Le code de la classe EquilateralTriangle, ci-dessous, montre comment la méthode `getArea()` :

```
package com.example.programmingas3.geometricshapes
{
    public class EquilateralTriangle extends RegularPolygon
    {
        public function EquilateralTriangle(len:Number = 100):void
        {
            super(len, 3);
        }

        public override function getArea():Number
        {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
        }

        public override function describe():String
        {
                /* starts with the name of the shape, then delegates the rest
                of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
        }
    }
}
```

Le mot-clé `override` indique que la méthode `EquilateralTriangle.getArea()` redéfinit volontairement la méthode `getArea()` de la superclasse RegularPolygon. Lorsque la méthode `EquilateralTriangle.getArea()` est appelée, elle calcule la surface à l'aide de la formule du code précédent. Le code de la méthode `RegularPolygon.getArea()` n'est jamais exécuté.

Par contre, la classe EquilateralTriangle ne définit pas sa propre version de la méthode`getPerimeter()`. Lorsque la méthode `EquilateralTriangle.getPerimeter()` est appelée, l'appel remonte la chaîne d'héritage et exécute le code de la méthode `getPerimeter()` de la superclasse RegularPolygon.

Le constructeur `EquilateralTriangle()` utilise l'instruction `super()` pour invoquer explicitement le constructeur `RegularPolygon()` de sa superclasse. Si les deux constructeurs avaient le même ensemble de paramètres, il serait possible d'omettre complètement le constructeur `EquilateralTriangle` et il suffirait d'exécuter le constructeur `RegularPolygon()`. Toutefois, le constructeur `RegularPolygon()` nécessite un paramètre supplémentaire, `numSides`. Le constructeur `EquilateralTriangle()` appelle donc `super(len, 3)` qui transmet le paramètre en entrée `len` et la valeur 3 pour indiquer que le triangle possède trois côtés.

La méthode `describe()` utilise également l'instruction `super()`, mais de façon différente, afin d'appeler la version de la méthode `describe()` figurant dans la superclasse RegularPolygon. La méthode `EquilateralTriangle.describe()` définit d'abord la variable de chaîne `desc` qui affiche le type de forme. Elle obtient ensuite les résultats de la méthode `RegularPolygon.describe()` en appelant `super.describe()` et elle ajoute ces résultats à la fin de la chaîne `desc`.

La classe Square n'est pas décrite en détail ici, mais elle est similaire à la classe EquilateralTriangle, avec un constructeur et ses propres implémentations des méthodes `getArea()` et `describe()`.

Polymorphisme et méthode usine (Factory)

Il est possible d'utiliser de diverses façons intéressantes un ensemble de classes qui utilisent à bon escient les interfaces et l'héritage. Par exemple, toutes les classes de formes décrites jusqu'ici implémentent l'interface IGeometricShape ou étendent une superclasse qui se charge de cette implémentation. Si vous définissez une variable comme étant une occurrence de IGeometricShape, il n'est pas nécessaire, pour appeler sa méthode `describe()`, de savoir si c'est une occurrence de la classe Circle ou de la classe Square.

Le code suivant illustre cette situation :

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

Lorsque `myShape.describe()` est appelée, elle exécute la méthode `Circle.describe()`; car bien que la variable soit définie comme une occurrence de l'interface IGeometricShape, Circle est sa classe sous-jacente.

Cet exemple illustre le principe du polymorphisme : le même appel à une méthode provoquera l'exécution d'un code différent, selon la classe de l'objet dont la méthode est appelée.

L'application GeometricShapes applique ce type de polymorphisme basé sur l'interface à l'aide d'une version simplifiée d'un modèle de conception appelé méthode usine (Factory). L'expression *méthode usine (Factory)* désigne une fonction qui renvoie un objet dont le type de données sous-jacent ou le contenu diffèrent selon le contexte.

La classe GeometricShapeFactory représentée ici définit une méthode usine (Factory) appelée `createShape()` :

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

La méthode usine (Factory) `createShape()` laisse aux constructeurs de la sous-classe de la forme le soin de définir les détails des occurrences qu'ils créent, tout en renvoyant les nouveaux objets comme occurrences de IgeometricShape, ce qui permet à l'application de les traiter de façon plus générale.

La méthode `describeShape()` de l'exemple précédent montre comment une application peut utiliser la méthode usine (Factory) pour obtenir une référence générique à un objet spécifique. L'application peut obtenir la description d'un objet Circle nouvellement créé en procédant comme suit :

```
GeometricShapeFactory.describeShape("Circle", 100);
```

La méthode `describeShape()` appelle alors la méthode usine (Factory) `createShape()` avec les mêmes paramètres. Elle met le nouvel objet Circle dans une variable statique appelée `currentShape`, dont le type a été défini comme objet de IGeometricShape. La méthode `describe()` est ensuite appelée pour l'objet `currentShape` et cet appel est automatiquement résolu pour exécuter la méthode `Circle.describe()` qui renvoie une description détaillée du cercle.

Amélioration de l'application d'exemple

La puissance des interfaces et de l'héritage devient évidente dès qu'il s'agit de modifier l'application.

Supposons que nous voulions ajouter une nouvelle forme, un pentagone, à cette application. Il suffit de créer une classe, Pentagon, qui étend la classe RegularPolygon et définit ses propres versions des méthodes `getArea()` et `describe()`. Une nouvelle option Pentagon est ensuite ajoutée dans l'interface utilisateur de l'application. C'est tout. La classe Pentagon obtiendra automatiquement, par héritage, les fonctionnalités des méthodes `getPerimeter()` et `getSumOfAngles()` de la classe RegularPolygon. Et puisqu'elle hérite d'une classe qui implémente l'interface IGeometricShape, une occurrence de Pentagon sera également traitée comme une occurrence de IGeometricShape. Autrement dit, il n'est pas nécessaire de modifier l'une des méthodes de la classe GeometricShapeFactory, ce qui rend beaucoup plus facile l'ajout ultérieur de nouveaux types de forme. Par conséquent, il n'est pas nécessaire non plus de modifier le code qui utilise la classe GeometricShapeFactory.

A titre d'exercice, il est conseillé d'ajouter une classe Pentagon à l'exemple Geometric Shapes, afin de constater à quel point les interfaces et l'héritage facilitent le processus d'ajout de nouvelles fonctionnalités à une application.