8

Programmation orientée objet

Concevoir autrement

La programmation orientée objet doit être considérée comme une manière de penser et de concevoir une application, certains la considère d'ailleurs comme un *paradigme de programmation*. Cette

nouvelle manière de penser n'est pas limitée à ActionScript 3 et peut être appliquée à un grand nombre de langages. Avant de démarrer, nous allons tout d'abord nous familiariser avec le vocabulaire puis nous entamerons notre aventure orientée objet à travers différents exemples concrets.

C'est en 1967 que le premier langage orienté objet vu le jour sous le nom de Simula-67. Comme son nom l'indique, Simula permettait de simuler toutes sortes de situations pour les applications de l'époque telles les applications de gestions, de comptabilité ou autres. Nous devons la notion d'objets, de classes, de sous-classes, d'événements ainsi que de ramasse-miettes (garbage collector) à Simula. Alan Key, inventeur du terme *programmation orientée objet* et ingénieur chez Xerox étendit les concepts apportés par Simula et développa le langage Smalltalk, aujourd'hui considéré comme le réel point de départ de la programmation orientée objet.

L'intérêt de la programmation orientée objet réside dans la *séparation des tâches*, *l'organisation* et la *réutilisation du code*. Nous pourrions résumer ce chapitre en une seule phrase essentielle à la base du concept de la programmation orienté objet : *A chaque type d'objet une tâche spécifique*.

Afin de développer un programme ActionScript nous avons le choix entre deux approches. La première, appelée couramment programmation procédurale ou fonctionnelle ne définit aucune séparation claire des tâches. Les différentes fonctionnalités de l'application ne sont pas affectées à un *objet spécifique*, des fonctions sont déclenchées dans un ordre précis et s'assurent que l'application s'exécute correctement.

En résumé, il s'agit de la première approche de programmation que tout développeur chevronné a connue. Au moindre bug ou mise à jour, c'est l'angoisse, aucun moyen d'isoler les comportements et les bogues, bienvenue dans le *code spaghetti*.

La seconde approche, appelée programmation orientée objet définit une séparation claire des tâches. Chaque objet s'occupe d'une tâche spécifique, dans le cas d'une galerie photo un objet peut gérer la connexion au serveur afin de récupérer les données, un autre s'occupe de l'affichage, enfin un dernier objet traite les entrées souris et clavier de l'utilisateur. En connectant ces objets entre eux, nous donnons vie à une application. Nous verrons plus tard que toute la difficulté de la programmation orientée objet, réside dans la communication inter objets.

La figure 8-1 illustre un exemple de galerie photo composée de trois objets principaux.

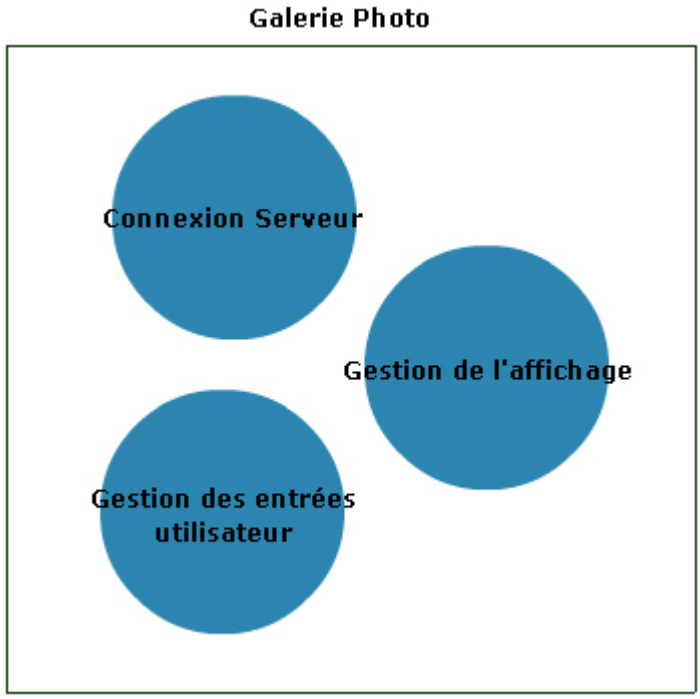

Figure 8-1. Concept de séparation des tâches.

Même si cela nous parait naturel, la notion de programmation orientée objet reste abstraite sans un cas concret appliqué aux développements de tous les jours. Afin de simplifier tout cela, nous pouvons regarder les objets qui nous entourent. Nous retrouvons à peu près partout le concept de *séparation des tâches*.

Afin d'optimiser le temps de développement et de maintenance, un programme peut être considéré comme un ensemble d'objets communicants pouvant à tout moment être remplacés ou réutilisés. Une simple voiture en est l'exemple parfait.

Le moteur, les roues, l'embrayage, les pédales, permettent à notre voiture de fonctionner. Si celle-ci tombe en panne, chaque partie sera testée, au cas où l'une d'entre elles s'avère défectueuse, elle est aussitôt remplacée ou réparée permettant une réparation rapide de la voiture. En programmation, la situation reste la même, si notre application objet est boguée, grâce à la séparation des tâches nous pouvons très facilement isoler les erreurs et corriger l'application ou encore ajouter de nouvelles fonctionnalités en ajoutant de nouveaux objets.

Tout est objet

Dans le monde qui nous entoure tout peut être représenté sous forme d'objet, une ampoule, une voiture, un arbre, la terre elle-même peut être considérée comme un objet.

En programmation, il faut considérer un objet comme une entité ayant un état et permettant d'effectuer des opérations. Pour simplifier nous pouvons dire qu'un objet est capable d'effectuer certaines tâches et possède des caractéristiques spécifiques.

Une télévision peut être considérée comme un objet, en possédant une taille, une couleur, ainsi que des fonctionnalités, comme le fait d'être allumée, éteinte ou en veille. Ici encore, la séparation des tâches est utilisée, le tube cathodique s'occupe de créer l'image, l'écran se charge de l'afficher, le capteur s'occupe de gérer les informations envoyées par la télécommande. Cet ensemble d'objets produit un résultat fonctionnel et utilisable. Nous verrons que de la même manière nous pouvons concevoir une application ActionScript 3 en associant chaque objet à une tâche spécifique.

Afin de porter notre exemple de télévision en ActionScript 3, nous pourrions définir une classe `Television` permettant de créer des objets télévisions. Chaque objet télévision serait donc de type `Television`. Lorsque un développeur parle de la classe `Array` nous savons de quel type d'objet il s'agit et de ses capacités, comme ajouter, supprimer ou trier des éléments. De la même manière, en parlant de la classe `Television` nous savons quelles sont les fonctionnalités offertes par ce type.

Si nous regardons à l'intérieur d'une télévision, nous y trouvons toutes sortes de composants. Des hauts parleurs, des boutons, ainsi qu'un ensemble de composants électroniques. En programmation, nous pouvons exactement reproduire cette conception, nous verrons que plusieurs objets travaillant ensemble peuvent donner naissance à une application concrète. Chaque composant peut ainsi être réutilisé dans un autre programme, les hauts parleurs d'un modèle de télévision pourront être réutilisés dans une autre.

C'est ce que nous appelons la notion de *composition*, nous reviendrons sur ce sujet au cours du chapitre 10 intitulé *Diffusion d'événements personnalisés*.

Une fois notre objet crée, nous pouvons lui demander d'exécuter certaines actions, lorsque nous achetons une télévision nous la choisissons selon ses capacités, et nous nous attentons au moins à pouvoir l'allumer, l'éteindre, monter le son ou baisser le son. Ces capacités propres à un objet sont définies par son *interface*, les

capacités de celle-ci sont directement liées au type. La figure 8-2 illustre l'exemple d'une télévision.

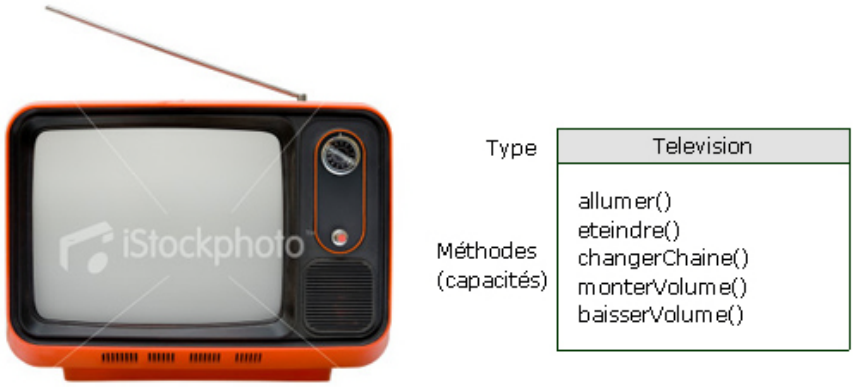

Figure 8-2. Classe `Television`.

Afin d'instancier un objet à partir d'une classe nous utilisons le mot clé `new` :

```
// création d'une instance de la classe Television au sein de la variable maTV
var maTV:Television = new Television();
```

Le type `Television` détermine l'interface de notre télévision, c'est-à-dire ce que l'objet est capable de faire. Les fonctions `allumer`, `eteindre`, `monterVolume`, et `baisserVolume` sont appelées méthodes car elles sont rattachées à un objet et définissent ses capacités. En les appelants, nous exécutons tout un ensemble de mécanismes totalement transparents pour l'utilisateur. Dans l'exemple suivant, nous instancions une télévision, s'il est trop tard nous l'éteignons :

```
// création d'une instance de la classe Television au sein de la variable maTV
var maTV:Television = new Television();

// nous allumons la télé
maTV.allumer()

// si il est trop tard
if ( heureActuelle > 12 )

{

    // nous étéignons la télé
    maTV.eteindre();

}
```

Comment stockons-nous les informations telles la taille, la couleur, ou bien le numéro de série de la télévision ?

Bonne question, ces données représentent les caractéristiques d'un objet et sont stockées au sein de *propriétés*. Nous pouvons imaginer qu'une télévision ait une taille, un poids, une couleur, et un numéro de série.

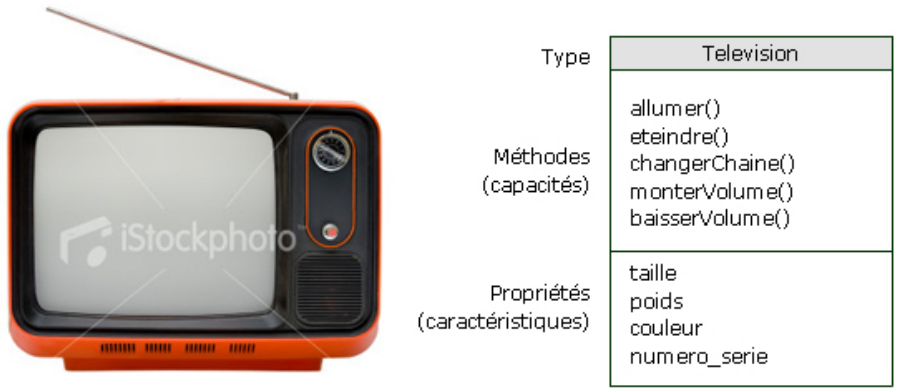

Figure 8-3. Méthodes et propriétés du type `Television`.

Toutes les instances de télévisions crées à partir de la même classe possèdent ainsi les même fonctionnalités, mais des caractéristiques différentes comme la couleur, la taille ou encore le poids. Pour résumer, nous pouvons dire que les méthodes et propriétés déterminent le type.

Pour récupérer la couleur ou le poids d'une télévision, nous ciblons la propriété voulue :

```
// nous récupérons la taille
var taille:Number = maTV.taille;

// nous récupérons la couleur
var couleur:Number = maTV.couleur;
```

Au sein de Flash nous retrouvons partout le concept d'objets, prenons le cas de la classe `MovieClip`, dans le code suivant nous créons une instance de `MovieClip` :

```
// instanciation d'un clip
var monClip:MovieClip = new MovieClip();
```

Nous savons d'après le type `MovieClip` les fonctionnalités disponibles sur une instance de `MovieClip` :

```
// méthodes de la classe MovieClip
monClip.gotoAndPlay(2);
monClip.startDrag();
```

En créant plusieurs instances de `MovieClip`, nous obtenons des objets ayant les mêmes capacités mais des caractéristiques différentes comme par exemple la taille, la position ou encore la transparence :

```
// instanciation d'un premier clip
var monPremierClip:MovieClip = new MovieClip();

// utilisation de l'api de dessin
monPremierClip.graphics.lineStyle ( 1 );
monPremierClip.graphics.beginFill ( 0x880099, 1);
monPremierClip.graphics.drawCircle ( 30, 30, 30 );

// caractéristiques du premier clip
monPremierClip.scaleX = .8;
monPremierClip.scaleY = .8;
monPremierClip.alpha = .5;
monPremierClip.x = 150;
monPremierClip.y = 150;

// affichage du premier clip
addChild ( monPremierClip );

// instanciation d'un second clip
var monSecondClip:MovieClip = new MovieClip();

// utilisation de l'api de dessin
monSecondClip.graphics.lineStyle ( 1 );
monSecondClip.graphics.beginFill ( 0x997711, 1);
monSecondClip.graphics.drawCircle ( 60, 60, 60 );

// caractéristiques du second clip
monSecondClip.scaleX = .8;
monSecondClip.scaleY = .8;
monSecondClip.alpha = 1;
monSecondClip.x = 300;
monSecondClip.y = 190;

// affichage du second clip
addChild ( monSecondClip );
```

Si nous testons le code précédent nous obtenons deux clips forme circulaire ayant des fonctionnalités communes mais des caractéristiques différentes, comme l'illustre la figure 8-4 :

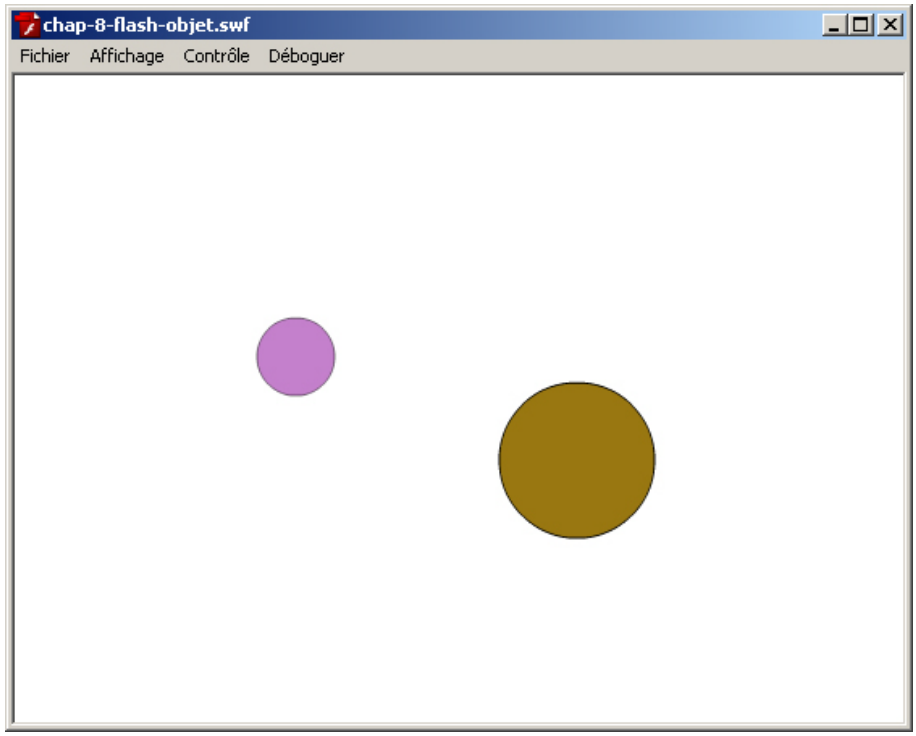

Figure 8-4. Deux clips de caractéristiques différentes.

Au sein de Flash tout est objet, toutes les classes permettent d'instancier des objets ayant des fonctionnalités spécifiques comme la gestion de l'affichage, le chargement de données externe, etc.

Nous allons à présent découvrir comment concevoir nos propres objets en ActionScript 3 afin de les rendre réutilisable et facile d'utilisation. Dans un nouveau fichier ActionScript nous allons créer une classe `Joueur` qui nous permettra de représenter un joueur dans un jeu. En quelques minutes notre classe sera définie et prête à être utilisée.

A retenir

- Tout est objet.
- A chaque objet une tâche spécifique.
- Les méthodes définissent les capacités d'un objet.
- Les propriétés définissent ses caractéristiques.
- Une instance de classe est un objet crée à partir d'une classe.
- Pour instancier un objet à partir d'une classe, nous utilisons le mot clé `new`.

Notre première classe

Pour créer un nouveau type d'objet en ActionScript 3 nous devons créer une classe. Il faut considérer celle-ci comme un moule permettant de créer des objets de même type. A partir d'une classe nous pouvons créer autant d'instances de cette classe que nous voulons.

Avant de commencer à coder, il est important de noter que les classes ActionScript 3 résident dans des fichiers externes portant l'extension `.as`. A coté de notre document Flash nous allons définir une classe ActionScript 3 avec un éditeur tel FlashDevelop ou Eclipse. Pour cela nous utilisons le mot clé `class` :

```
class Joueur

{



}
```

Nous sauvons cette classe à coté de notre document en cours et nous l'appelons `Joueur.as`. Sur le scénario principal nous instancions notre joueur avec le mot clé `new` :

```
var monJoueur:Joueur = new Joueur();
```

A la compilation nous obtenons les deux erreurs suivantes :

```
1046: Ce type est introuvable ou n'est pas une constante de compilation :
Joueur.
1180: Appel à une méthode qui ne semble pas définie, Joueur.
```

Le compilateur ne semble pas apprécier notre classe, il ne reconnaît pas cette classe que nous venons de définir. Contrairement à ActionScript 2, en ActionScript 3, une classe doit être contenue dans un conteneur de classes appelé paquetage.

Introduction aux paquetages

Les paquetages (*packages* en anglais) permettent dans un premier temps d'organiser les classes en indiquant dans quel répertoire est située une classe. Nous pouvons imaginer une application dynamique dans laquelle une classe `Connector` serait placée dans un répertoire `serveur`, une autre classe `PNGEncoder` serait elle placée dans un répertoire `encodage`.

Le compilateur trouvera la classe selon le chemin renseigné par le paquetage, afin de définir un paquetage nous utilisons le mot clé `package`.

Si notre classe `Joueur` était placée dans un répertoire `jeu` nous devrions définir notre classe de la manière suivante :

```
 package jeu

{
    public class Joueur
    {


    }

}
```

Il serait alors nécessaire de spécifier son chemin au compilateur en utilisant l'instruction `import` avant de pouvoir l'utiliser :

```
import jeu.Joueur;
```

Lorsque notre classe n'est pas placée dans un répertoire spécifique mais réside simplement à coté de notre document FLA aucun import n'est nécessaire, le compilateur vérifie automatiquement à côté du document Flash si les classes existent.

Dans notre cas, la classe `Joueur` est placée à coté du document Flash et ne réside dans aucun répertoire, nous devons donc spécifier un paquetage vide :

```
package

{
    class Joueur
    {


    }

}
```

Dans ce cas, aucun import préalable n'est nécessaire. En ActionScript 2, le mot clé `package` n'existait pas, pour indiquer le chemin d'accès

à une classe nous spécifions le chemin d'accès directement lors de sa déclaration :

```
class jeu.Joueur
{


}
```

La notion de paquetages an ActionScript 3 ne sert pas uniquement à refléter l'emplacement d'une classe, en ActionScript 2, une seule classe pouvait être définie au sein d'un fichier source. En résumé, une seule et unique classe pour chaque fichier .as.

En ActionScript 3, quelques subtilités on été ajoutées, nous pouvons désormais définir autant de classes que nous souhaitons pour chaque fichier .as. Une seule classe résidera à l'intérieur d'une déclaration de paquetage, les autres devront être définies à l'extérieur de celui-ci et ne pourront pas être utilisées par un code extérieur. Ne vous inquiétez pas, cela parait relativement abstrait au départ, nous allons au cours des prochains chapitres apprendre à maîtriser la notion de paquetages et découvrir à nouveau de nouvelles fonctionnalités.

Si nous tentons de compiler notre classe à nouveau, les mêmes erreurs s'affichent. Afin de pouvoir instancier notre classe depuis l'extérieur nous devons utiliser l'attribut de classe `public`. Nous allons revenir sur la notion d'attributs de classe très bientôt :

```
package

{
    public class Joueur
    {


    }

}
```

Une fois le mot clé `public` ajouté, la compilation s'effectue sans problème. L'intérêt de notre classe est de pouvoir être instanciée et recevoir différents paramètres lorsque nous utiliserons cette classe dans nos applications. Il y'a de fortes chances qu'un joueur ait un nom, un prénom, ainsi que d'autres propriétés.

Pour instancier notre instance de `Joueur` nous utilisons le mot clé `new` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur();
```

Notre objet `Joueur` est crée pour savoir si un objet correspond à un type spécifique nous utilisons le mot clé `is` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur ();

// est-ce que notre objet est de type Joueur ?
// affiche : true
trace( monJoueur is Joueur );
```

Le mot clé `instanceof` utilisé en ActionScript 1 et 2 n'existe plus et a été remplacé par le mot clé `is`. En étant de type `Joueur` notre instance nous garantit qu'elle possède toutes les capacités et propriétés propres au type `Joueur`.

A retenir :

- Pour définir une classe, nous utilisons le mot clé `class`.
- Les classes ActionScript 3 résident dans des fichiers externes portant l'extension .as.
- Les classes ActionScript 3 doivent résider dans des paquetages
- Le nom de la classe doit être le même que le fichier .as
- Afin d'utiliser une classe nous devons l'importer avec la mot clé `import`.
- Afin de définir un paquetage, nous utilisons le mot clé `package`.
- Pour tester le type d'un objet nous utilisons le mot clé `is`.

Définition de propriétés

Comme nous l'avons vu précédemment, les propriétés d'un objet décrivent ses caractéristiques. Un être humain peut être considéré comme une instance de la classe `Humain` où chaque caractéristique propre à l'être humain telle la taille, la couleur de ses yeux, ou son nom seraient stockées au sein de propriétés.

Nous allons définir des propriétés au sein de la classe `Joueur` afin que tout les joueurs crées aient leur propres caractéristiques. Chaque joueur aura un nom, un prénom, une ville d'origine, un âge, et un identifiant les caractérisant. Nous définissons donc cinq propriétés au sein de la classe `Joueur`.

Pour définir des propriétés au sein d'une classe, nous utilisons la définition de classe, un espace situé juste en dessous de la ligne définissant la classe :

```
package

{
    public class Joueur
    {

        // définition des propriétés de la classe
        var nom:String;
```

```
            var prenom:String;
            var age:int;
            var ville:String;

        }

    }
```

En lisant les lignes précédentes nous nous rendons compte que la syntaxe nous rappelle la définition de simples variables.

> Les propriétés sont en réalité des variables comme les autres, mais évoluant dans le contexte d'un objet.

En définissant ces propriétés, nous garantissons que chaque instance de la classe `Joueur` les possède. A tout moment, nous pouvons récupérer ces informations en ciblant les propriétés sur l'instance de classe :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur();

// recupération du prénom du joueur
var prenom = monJoueur.prenom;
```

A la compilation le code précédent échoue, car contrairement à ActionScript 2, lorsque nous ne spécifions pas d'attributs pour chaque propriété, celles-ci sont considérées comme non visible depuis l'extérieur du paquetage en cours. Pour gérer l'accès aux propriétés d'une classe nous utilisons les attributs de propriétés.

Attributs de propriétés de classe

Nous définissons comme membres d'une classe, les propriétés ainsi que les méthodes. Afin de gérer l'accès à chaque membre, nous utilisons des attributs de propriétés.

> Au sein du modèle d'objet ActionScript les méthodes sont aussi définies par des propriétés. Une méthode n'étant qu'une fonction évoluant dans le contexte d'un objet.

Cinq attributs de propriétés de classe existent en ActionScript 3 :

- `internal` : Par défaut, le membre est accessible uniquement depuis le paquetage en cours.
- `public` : Accessible depuis n'importe quelle partie du code.
- `private` : Le membre n'est accessible que depuis la même classe. Les sous-classes n'ont pas accès au membre.
- `protected` : Le membre est accessible depuis la même classe, et les sous-classes.

- `static` : Le membre est accessible uniquement depuis la classe, non depuis les instances.

Nous n'avons pas défini pour le moment d'attributs pour les propriétés de notre classe `Joueur`, lorsqu'aucun attribut n'est défini, la propriété ou méthode est considérée comme `internal` et n'est accessible que depuis une classe appartenant au même paquetage.

Nous reviendrons sur ces subtilités plus tard, pour le moment nous souhaitons accéder à nos propriétés depuis notre scénario nous les définissons comme `public` :

```
package

{
    public class Joueur
    {

        // définition des propriétés publiques de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;

    }

}
```

C'est pour cette raison que nous avons défini notre classe publique afin que celle-ci puisse être instanciée depuis l'extérieur.

Une fois nos propriétés rendues publiques nous pouvons y accéder depuis l'extérieur, nous instancions un objet `Joueur` puis nous accédons à sa propriété `prenom` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur();

// recupération du prénom du joueur
var prenom = monJoueur.prenom;

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

En ciblant la propriété `prenom` d'un joueur nous récupérons la valeur `null`. Ce comportement est tout à fait normal, car pour l'instant aucunes données ne sont contenues par les propriétés. Nous les avons simplement définies.

> En ActionScript 2 le code précédent aurait retourné `undefined` au lieu de `null`.

Nous venons de voir qu'à l'aide d'attributs de propriétés de classe nous pouvons gérer l'accès et le comportement des membres d'une

classe. Il existe en ActionScript 3 d'autres attributs liés cette fois aux classes, ces derniers sont appelés *attributs de classe*.

Attributs de classe

Certaines règles peuvent être appliquées aux classes directement à l'aide de quatre attributs, notons que l'attribut `abstract` n'existe pas en ActionScript 3 :

- `dynamic` : La classe accepte l'ajout de propriétés ou méthodes à l'exécution.
- `final` : La classe ne peut pas être étendue.
- `internal` (par défaut) : La classe n'est accessible que depuis les classes appartenant au même paquetage.
- `public` : La classe est accessible depuis n'importe quel emplacement.

Afin de rendre la classe `Joueur` instanciable depuis l'extérieur nous avons du la rendre publique à l'aide de l'attribut `public`.

Celle-ci est d'ailleurs considérée comme non dynamique, cela signifie qu'il est impossible d'ajouter à l'exécution de nouvelles méthodes ou propriétés à une occurrence de la classe ou à la classe même.

Imaginons que nous souhaitions ajouter une nouvelle propriété appelée `sSexe` à l'exécution :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ();

// ajout d'une nouvelle propriété à l'exécution
premierJoueur.sSexe = "H";
```

Le compilateur se plaint et génère l'erreur suivante :

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

Si nous rendons la classe dynamique à l'aide de l'attribut `dynamic`, l'ajout de membres à l'exécution est possible :

```
package

{
    dynamic public class Joueur
    {

        // définition des propriétés publiques de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;

    }
```

```
}
```

Dans le code suivant nous créons une nouvelle propriété `sSexe` au sein l'instance de classe `Joueur` et récupérons sa valeur :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ();

// ajout d'une nouvelle propriété à l'exécution
premierJoueur.sSexe = "H";

// récupération de la valeur
// affiche : H
trace( premierJoueur.sSexe );
```

Bien que cela soit possible, il est fortement déconseillé d'ajouter de nouveaux membres à l'exécution à une classe ou instance de classe, la lecture de la classe ne reflètera pas les réelles capacités ou caractéristiques de la classe.

En lisant la définition de la classe `Joueur` le développeur pensera trouver cinq propriétés, en réalité une sixième est rajoutée à l'exécution. Ce dernier serait obligé de parcourir tout le code de l'application afin de dénicher toutes les éventuelles modifications apportées à la classe à l'exécution.

Afin d'initialiser notre objet `Joueur` lors de sa création nous devons lui passer des paramètres. Même si notre objet `Joueur` est bien crée, son intérêt pour le moment reste limité car nous ne passons aucun paramètre lors son instanciation. Pour passer des paramètres à un objet lors de son instanciation nous définissons une méthode au sein de la classe appelée *méthode constructeur*.

Le constructeur

Le but du constructeur est d'initialiser l'objet crée. Celui-ci sera appelé automatiquement dès que la classe sera instanciée.

> Attention, le constructeur doit impérativement avoir le même nom que la classe. Si ce n'est pas le cas, il sera considéré comme une méthode classique et ne sera pas déclenché.

Nous allons prévoir quatre paramètres d'initialisation pour chaque joueur :

- Nom : Une chaine de caractères représentant son nom.
- Prénom : Une chaine de caractères représentant son prénom.
- Age : Un entier représentant son âge.
- Ville : Une chaine de caractères représentant sa location.

Au sein du constructeur nous définissons quatre paramètres, qui serviront à accueillir les futurs paramètres passés :

```
package

{
    public class Joueur
    {

        // définition des propriétés de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

        {

            trace( this );

        }

    }

}
```

Si nous tentons d'instancier notre joueur sans passer les paramètres nécessaires :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur();
```

Le compilateur nous renvoie une erreur :

```
1136: Nombre d'arguments incorrect.  4 attendus.
```

Le compilateur ActionScript 2 ne nous renvoyait aucune erreur lorsque nous tentions d'instancier une classe sans paramètres alors que celle-ci en nécessitait. ActionScript 3 est plus strict et ne permet pas l'instanciation de classes sans paramètres si cela n'est pas spécifié à l'aide du mot clé `rest`. Nous verrons plus tard comment définir une classe pouvant recevoir ou non des paramètres à l'initialisation.

Pour obtenir un joueur, nous créons une instance de la classe `Joueur` en passant les paramètres nécessaires :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// recupération du prénom du joueur
var prenom = monJoueur.prenom;

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

Si nous tentons d'instancier un joueur en passant plus de paramètres que prévus :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan", 50);
```

L'erreur de compilation suivante est générée :

```
1137: Nombre d'arguments incorrect.  Nombre maximum attendu : 4.
```

Le compilateur ActionScript 2 était moins strict et permettait de passer un nombre de paramètres en plus de ceux présents au sein de la signature d'une méthode, en ActionScript 3 cela est impossible

Nous avons instancié notre premier joueur, mais lorsque nous tentons de récupérer son prénom, la propriété `prenom` nous renvoie à nouveau `null`. Nous avons bien passé les paramètres au constructeur, et pourtant nous récupérons toujours `null` lorsque nous ciblons la propriété `prenom`. Est-ce bien normal ?

C'est tout à fait normal, il ne suffit pas de simplement définir un constructeur pour que l'initialisation se fasse comme par magie. C'est à nous de gérer cela, et d'intégrer au sein du constructeur une affectation des paramètres aux propriétés correspondantes. Une fois les paramètres passés, nous devons les stocker au sein de l'objet afin d'être mémorisées, c'est le travail du constructeur, ce dernier doit recevoir les paramètres et les affecter à des propriétés correspondantes définies au préalable.

Pour cela nous modifions le constructeur de manière à affecter chaque paramètre passé aux propriétés de l'objet :

```
package

{
    public class Joueur
    {

          // définition des propriétés de la classe
          public var nom:String;
          public var prenom:String;
          public var age:int;
          public var ville:String;
          public var id:int;

          // fonction constructeur
          function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

          {

               // affectation de chaque propriété
               prenom = pPrenom;
               nom = pNom;
               age = pAge;
               ville = pVille;
```

```
            }

        }

    }
```

Une fois les propriétés affectées, nous pouvons récupérer les valeurs associées :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// récupération du prénom du joueur
var prenom:String = monJoueur.prenom;
var nom:String = monJoueur.nom;
var age:int = monJoueur.age;
var ville:String = monJoueur.ville;

// affiche : Stevie
trace( prenom );
// affiche : Wonder
trace( nom );
// affiche : 57
trace( age );
// affiche : Michigan
trace( ville );
```

Lorsqu'un objet `Joueur` est instancié nous passons des paramètres d'initialisation, chaque caractéristique est stockée au sein de propriétés.

A l'inverse si nous rendons ces propriétés privées :

```
package

{
    public class Joueur
    {

        // définition des propriétés de la classe
        private var nom:String;
        private var prenom:String;
        private var age:int;
        private var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

        {

            prenom = pPrenom;
            nom = pNom;
            age = pAge;
            ville = pVille;

        }

    }
```

```
}
```

En définissant l'attribut `private`, les propriétés deviennent inaccessibles depuis l'extérieur de la classe :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affiche une erreur à la compilation :
// 1178: Tentative d'accès à la propriété inaccessible nom, via la référence de
type static Joueur.
trace( monJoueur.nom );
```

Lorsque le compilateur est en mode strict, une erreur à la compilation est levée, il est impossible de compiler.

Si nous passons le compilateur en mode non strict, à l'aide de la syntaxe crochet il devient possible de compiler :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");

// provoque une erreur à l'exécution :
trace( monJoueur['nom'] );
```

Mais la machine virtuelle 2 nous rattrape en levant une erreur à l'exécution :

```
ReferenceError: Error #1069: La propriété nom est introuvable sur Joueur et il
n'existe pas de valeur par défaut.
```

Souvenons-nous que la machine virtuelle d'ActionScript 3 (VM2) conserve les types à l'exécution. L'astuce ActionScript 2 consistant à utiliser la syntaxe crochet pour éviter les erreurs de compilation n'est plus valable. Nous verrons tout au long des prochains chapitres, différents cas d'utilisation d'autres attributs de propriétés.

Mais quel serait l'intérêt de définir des propriétés privées ? Quel est l'intérêt si nous ne pouvons pas y accéder ?

Nous allons découvrir très vite d'autres moyens plus sécurisés d'accéder aux propriétés d'un objet. Nous allons traiter cela plus en détail dans quelques instants. Nous venons de découvrir comment définir des propriétés au sein d'une classe ActionScript 3, abordons maintenant la définition de méthodes.

A retenir :

- Le constructeur d'une classe sert à recevoir des paramètres afin d'initialiser l'objet.
- La présence de constructeur n'est pas obligatoire, si omis, le compilateur en définit un par défaut. Dans ce cas, aucun paramètre ne peut être passé lors de l'instanciation de la classe.

Utilisation du mot clé this

Afin de faire référence à *l'objet courant* nous pouvons utiliser le mot clé `this` au sein d'une classe. Nous aurions pu définir la classe `Joueur` de la manière suivante :

```
package

{
    public class Joueur
    {

        // définition des propriétés de la classe
        public var nom:String;
        public var prenom:String;
        public var age:int;
        public var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

        {

            // affectation de chaque propriété
            this.prenom = pPrenom;
            this.nom = pNom;
            this.age = pAge;
            this.ville = pVille;

        }

    }

}
```

Bien que son utilisation soit intéressante dans certaines situations comme pour passer une référence, son utilisation peut s'avérer redondante et rendre le code verbeux. Certains développeurs apprécient tout de même son utilisation afin de différencier facilement les propriétés d'occurrences des variables locales.

En voyant une variable précédée du mot clé `this`, nous sommes assurés qu'il s'agit d'une propriété membre de la classe. En résumé, il s'agit d'un choix propre à chaque développeur, il n'existe pas de véritable règle d'utilisation.

Définition de méthodes

Alors que les propriétés d'un objet définissent ses caractéristiques, les méthodes d'une classe définissent ses capacités. Nous parlons généralement de fonctions membres. Dans notre précédent exemple de télévision, les méthodes existantes étaient `allumer`, `eteindre`, `changerChaine`, `baisserVolume`, et `monterVolume`.

En appliquant le même concept à notre classe `Joueur` il parait logique qu'un joueur puisse se présenter, nous allons donc définir une méthode `sePresenter`. Une méthode est un membre régit aussi par les attributs de propriétés. Nous allons rendre cette méthode publique car elle devra être appelée depuis l'extérieure :

```
package

{

    public class Joueur
    {

          // définition des propriétés de la classe
          public var nom:String;
          public var prenom:String;
          public var age:int;
          public var ville:String;

          // fonction constructeur
          function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

          {

               prenom = pPrenom;
               nom = pNom;
               age = pAge;
               ville = pVille;

          }

          // méthode permettant au joueur de se présenter
          public function sePresenter ( ):void

          {

               trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

          }

    }

}
```

Pour qu'un joueur se présente nous appelons la méthode `sePresenter` :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");
```

```
// Je m'appelle Stevie, j'ai 57 ans.
monJoueur.sePresenter();
```

Ainsi, chaque instance de `Joueur` a la capacité de se présenter. La méthode `sePresenter` récupère les propriétés `prenom` et `age` et affiche leurs valeurs. Toutes les propriétés définies au sein de la classe sont accessibles par toutes les méthodes de celle-ci.

Nous allons définir une deuxième méthode nous permettant d'afficher une représentation automatique de l'objet `Joueur`. Par défaut lorsque nous traçons un objet, le lecteur Flash représente l'objet sous une forme simplifiée, indiquant simplement le type de l'objet :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affiche : [object Joueur]
trace( monJoueur );
```

Notons qu'en ActionScript 2, le lecteur affichait simplement `[objet Object]` il fallait ensuite tester à l'aide de l'instruction `instanceof` afin de déterminer le type d'un objet. En ActionScript 3 l'affichage du type est automatique.

En définissant une méthode `toString` au sein de la classe `Joueur` nous redéfinissons la description que le lecteur fait de l'objet :

```
package

{

    public class Joueur
    {

          // définition des propriétés de la classe
          public var nom:String;
          public var prenom:String;
          public var age:int;
          public var ville:String;

          // fonction constructeur
          function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

          {

               prenom = pPrenom;
               nom = pNom;
               age = pAge;
               ville = pVille;

          }

          // méthode permettant au joueur de se présenter
          public function sePresenter ( ):void

          {
```

```
            trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

        }

        // affiche une description de l'objet Joueur
        public function toString ( ):String

        {

            return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";

        }

    }

}
```

La méthode `toString` retourne une chaine de caractères utilisée par le lecteur Flash lorsque celui-ci affiche la description d'un objet. Une fois définie, lorsque nous traçons une instance de la classe `Joueur`, nous obtenons une représentation détaillée de l'objet.

Cela rend notre code plus élégant, lorsqu'un développeur tiers cible une instance de la classe `Joueur`, la description suivante est faite :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affiche : [Joueur prenom : Stevie,  nom : Wonder, age : 57, ville :
Michigan]
trace( monJoueur );
```

Dans la quasi-totalité des classes que nous créeront, nous intégrerons une méthode `toString` afin d'assurer une description élégante de nos objets.

Pour le moment nous accédons directement depuis l'extérieur aux propriétés de l'objet `Joueur` car nous les avons définies comme publique. Afin de rendre notre conception plus solide nous allons aborder à présent un nouveau point essentiel de la programmation orienté objet appelé *encapsulation*.

A retenir :

- Les méthodes de la classe ont accès aux propriétés définies au sein de la classe.
- En définissant une méthode `toString` sur nos classes nous modifions la représentation de notre objet faite par le lecteur Flash.

L'encapsulation

Lorsque nous utilisons une télévision nous utilisons des fonctionnalités sans savoir ce qui se passe en interne, et tant mieux. L'objet nous est livré mettant à disposition des fonctionnalités, nous ne savons rien de ce qui se passe en interne.

Le terme d'encapsulation détermine la manière dont nous exposons les propriétés d'un objet envers le monde extérieur. Un objet bien pensé doit pouvoir être utilisé sans montrer les détails de son *implémentation*. En d'autres termes, un développeur tiers utilisant notre classe `Joueur` n'a pas à savoir que nous utilisons une propriété nommée `age` ou `prenom` ou autres.

Il faut tout d'abord séparer les personnes utilisant les classes en deux catégories :

- Le ou les auteurs de la classe.
- Les personnes l'utilisant.

Le code interne à une classe, est appelé `implémentation`. En tant qu'auteur de classes, nous devons impérativement nous assurer de ne pas montrer les détails de l'implémentation de notre classe. Pourquoi ?

Afin d'éviter que la modification d'un détail de l'implémentation n'entraine de lourdes répercussions pour les développeurs utilisant nos classes.

Prenons un scénario classique, Mathieu, Nicolas et Eric sont développeurs au sein d'une agence Web et développent différentes classes ActionScript 3 qu'ils réutilisent au cours des différents projets qu'ils doivent réaliser. Nicolas vient de développer une classe permettant de générer des PDF en ActionScript 3. Toute l'équipe de développement Flash utilise cette classe dans les nouveaux projets et il faut bien avouer que tout le monde est plutôt satisfait de cette nouvelle librairie.

La figure 8-4 illustre la classe que Nicolas a développée, ainsi que les différentes fonctionnalités disponibles :

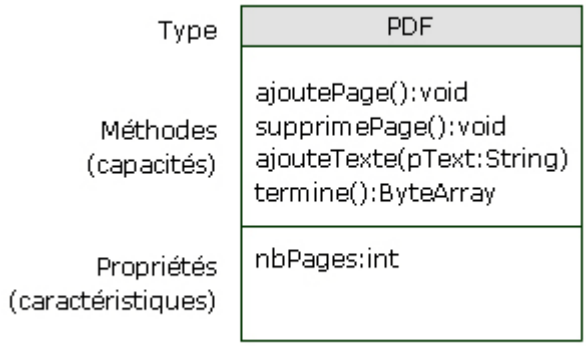

Figure 8-4. Schéma UML simplifié de la classe PDF.

La propriété `nbPages` permet de savoir combien de pages comporte le PDF en cours de création. A chaque page crée, Nicolas incrémente la valeur de cette propriété.

Eric vient justement de livrer un projet à un client qui souhaitait générer des PDF depuis Flash. Eric a utilisé la propriété `nbPages` afin de savoir combien de pages sont présentes dans le PDF en cours de création, voici une partie du code d'Eric :

```
// récupération du nombre de pages
var nombresPages:int = monPDF.nbPages;
// affichage du nombre de pages
legende_pages.text = String ( nombresPages );
```

Un matin, Nicolas, auteur de la classe se rend compte que la classe possède quelques faiblesses et décide de rajouter des fonctionnalités. Sa gestion des pages internes au document PDF ne lui parait pas optimisée, désormais Nicolas stocke les pages dans un tableau accessible par la propriété `tableauPages` contenant chaque page du PDF. Voici la nouvelle version de la classe PDF :

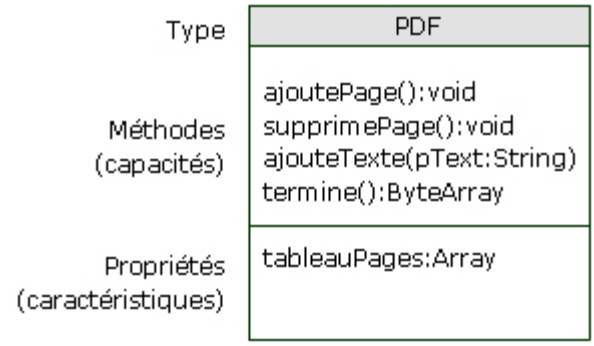

Figure 8-5. Nouvelle version de la classe PDF.

Désormais Nicolas utilise un tableau pour stocker chaque page crée. Pour récupérer le nombre de pages, il faut cibler le tableau `tableauPages` et récupérer sa longueur pour connaitre le nombre de pages du PDF. Eric et Mathieu récupèrent la nouvelle version mais tous leurs projets ne compilent plus, car la propriété `nbPages` n'existe plus !

Eric et Mathieu auront beau tenter de convaincre Nicolas de remettre la propriété afin que tout fonctionne à nouveau, Nicolas, très content de sa nouvelle gestion interne des pages refuse de rajouter cette propriété `nbPages` qui n'aurait plus de sens désormais.

Qui est le fautif dans ce scénario ?

Nicolas est fautif d'avoir exposé à l'extérieur cette propriété `nbPages`. En modifiant l'implémentation de sa classe, chaque développeur utilisant la classe risque de voir son code obsolète. Nicolas aurait du définir une méthode d'accès `getNbPages` qui peut importe l'implémentation aurait toujours retourné le nombre de pages. Nicolas se rendant compte de sa maladresse, décide de sortir une nouvelle version prenant en compte l'encapsulation. La figure 8-6 illustre la nouvelle classe.

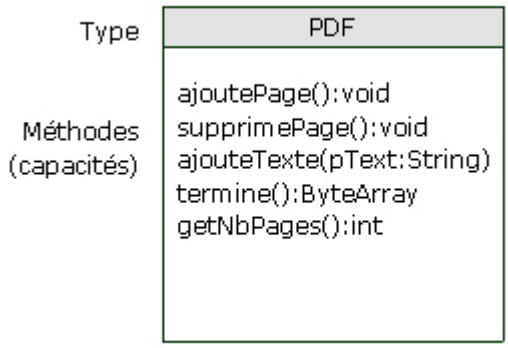

Figure 8-6. Nouvelle version encapsulée de la classe PDF.

Eric et Mathieu en appelant la méthode `getNbPages` ne savent pas ce qui se passe en interne et tant mieux ! Nicolas est tranquille et pourra modifier tout ce qu'il souhaite en interne, son seul soucis sera de toujours retourner le nombre de pages quelque soit l'implémentation.

> Ainsi nous pouvons modifier l'implémentation interne de l'objet sans modifier l'interface de programmation.

Nous allons mettre en pratique la notion d'encapsulation et découvrir d'autres avantages liés à ce nouveau concept.

Mise en pratique de l'encapsulation

Nous venons de voir qu'il est très important de cacher l'implémentation afin de rendre notre conception plus solide. Les parties cachées du code ne sont pas utilisées par les développeurs, ainsi nous sommes libres de le modifier sans entrainer de répercussions négatives. L'encapsulation des propriétés permet ainsi de rendre notre code plus intuitif, nous rendons visible uniquement ce qui est utile et utilisable.

Dans le cas de notre classe `Joueur` nous allons conserver la plupart des propriétés accessibles mais sous une forme différente. Pour accéder à l'âge ou au nom d'un joueur accédons directement à des propriétés publiques :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : Stevie
trace( monJoueur.prenom );
// affiche : 57
trace( monJoueur.age );
```

Si un développeur tiers vient à utiliser notre classe `Joueur`, ce dernier pourra utiliser ces deux propriétés. Si nous changeons plus tard l'implémentation de notre classe, ces propriétés pourraient peut être disparaitre, ou bien être renommées rendant le code du développeur caduque.

Afin d'éviter tout risque, nous allons définir des méthodes d'accès qui tâcheront de renvoyer la valeur escomptée, quelque soit l'implémentation.

Les méthodes d'accès

Deux groupes de méthodes d'accès existent :

- Les méthodes récupérant la valeur d'une propriété, appelées méthodes de récupération.
- Les méthodes affectant une valeur à une propriété, appelées méthodes d'affectation.

Prenons un exemple simple, au lieu de cibler directement la propriété `age`, le développeur tiers appellera une méthode `getAge`. Nous rendons dans un premier temps toutes nos propriétés privées afin que celles-ci soit cachées :

```
package
```

```
{

    public class Joueur
    {

          // définition des propriétés de la classe
          private var nom:String;
          private var prenom:String;
          private var age:int;
          private var ville:String;

          // fonction constructeur
          function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

          {

               prenom = pPrenom;
               nom = pNom;
               age = pAge;
               ville = pVille;

          }

          // méthode permettant au joueur de se présenter
          public function sePresenter ( ):void

          {

               trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

          }

          // affiche une description de l'objet Joueur
          public function toString ( ):String

          {

               return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";

          }

    }

}
```

En programmation orientée objet, il est fortement conseillé de protéger les propriétés en les rendant privées. Puis nous définissons une méthode `getAge` afin de récupérer l'âge du joueur :

```
package

{

    public class Joueur
    {

          // définition des propriétés de la classe
          private var nom:String;
          private var prenom:String;
```

```
        private var age:int;
        private var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

        {

             prenom = pPrenom;
             nom = pNom;
             age = pAge;
             ville = pVille;

        }

        // récupère l'age du joueur
        public function getAge ( ):int

        {

             return age;

        }

        // méthode permettanrt au joueur de se presenter
        public function sePresenter ( ):void

        {

             trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

        }

        // affiche une description de l'objet Joueur
        public function toString ( ):String

        {

             return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";

        }

    }

}
```

Si nous tentons d'accéder aux propriétés privées, le compilateur empêche de compiler :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : 1178: Tentative d'accès à la propriété inaccessible prenom, via la
référence de type static Joueur.
trace( monJoueur.prenom );

// affiche : 1178: Tentative d'accès à la propriété inaccessible age, via la
référence de type static Joueur.
trace( monJoueur.age );
```

Désormais, afin de récupérer l'âge d'un joueur nous appelons la méthode `getAge` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : 57
trace( monJoueur.getAge() );
```

En utilisant des méthodes d'accès nous cachons l'implémentation de la classe et rendons les modifications futures sécurisées et plus faciles. Les développeurs ne savent pas que la propriété `age` est utilisée en interne. Si nous décidons de changer l'implémentation pour des raisons d'optimisations ou de conception, aucun risque pour les développeurs utilisant la classe.

Dans l'exemple suivant les propriétés du joueur sont désormais stockées au sein d'un objet :

```
package

{

    public class Joueur
    {

          // définition des propriétés de la classe
          private var infosJoueur:Object;

          // fonction constructeur
          function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

          {

               // les paramètres sont désormais stockés
               // dans un objet infosJoueur
               infosJoueur = new Object();
               // chaque paramètre est stocké dans l'objet infoJoueurs
               infosJoueur.prenom = pPrenom;
               infosJoueur.nom = pNom;
               infosJoueur.age = pAge;
               infosJoueur.ville = pVille;

          }

          // récupère l'age du joueur
          public function getAge ( ):int

          {

               //récupère la propriété age au sein de l'objet infosJoueur
               return infosJoueur.age;

          }

          // méthode permettanrt au joueur de se presenter
          public function sePresenter ( ):void
```

```
        {

            trace("Je m'appelle " + infosJoueur.prenom + ", j'ai " +
infosJoueur.age + " ans." );

        }

        // affiche une description de l'objet Joueur
        public function toString ( ):String

        {

            return "[Joueur prenom : " + infosJoueur.prenom +",  nom : " +
infosJoueur.nom + ", age : " + infosJoueur.age + ", ville : " +
infosJoueur.ville + "]";

        }

    }

}
```

L'accès aux propriétés ne change pas, notre code continue de fonctionner :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affiche : 57
trace( monJoueur.getAge() );
```

Ainsi, notre implémentation change sans intervenir sur l'interface de programmation.

Il est important de signaler qu'une méthode de récupération peut porter n'importe quel nom, l'utilisation du mot `get` au sein de la méthode `getAge` n'est pas obligatoire. Nous aurions pu appeler cette même méthode `recupAge`.

Nous définissons une méthode de récupération spécifique à chaque propriété :

```
package

{

    public class Joueur
    {

        // définition des propriétés de la classe
        private var nom:String;
        private var prenom:String;
        private var age:int;
        private var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)
```

```
            {

                prenom = pPrenom;
                nom = pNom;
                age = pAge;
                ville = pVille;

            }

            // récupère le prénom du joueur
            public function getPrenom ( ):String

            {

                return prenom;

            }

            // récupère le nom du joueur
            public function getNom ( ):String

            {

                return nom;

            }

            // récupère l'age du joueur
            public function getAge ( ):int

            {

                return age;

            }

            // récupère la ville du joueur
            public function getVille ( ):String

            {

                return ville;

            }

            // méthode permettant au joueur de se presenter
            public function sePresenter ( ):void

            {

                trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

            }

            // affiche une description de l'objet Joueur
            public function toString ( ):String

            {

                return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";
```

```
        }

    }

}
```

De cette manière, nous pouvons accéder à toutes les propriétés d'une instance de `Joueur` :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affiche : Stevie
trace( monJoueur.getPrenom() );

// affiche : Wonder
trace( monJoueur.getNom() );

// affiche : 57
trace( monJoueur.getAge() );

// affiche : Michigan
trace( monJoueur.getVille() );
```

Grace aux différentes méthodes de récupération les propriétés privées sont rendues accessibles de manière encapsulée. Pour l'instant il est impossible de les modifier, car nous n'avons pas encore intégré de méthodes d'affectation.

Contrôle d'affectation

Nous venons de voir comment rendre notre code encapsulé grâce aux méthodes de récupération et d'affectation, l'autre grand intérêt de l'encapsulation concerne le contrôle d'affectation des propriétés.

Sans méthodes d'affectation, il est impossible de rendre l'affectation ou l'accès à nos propriétés intelligentes. Imaginons que le nom de la ville doive toujours être formaté correctement. Pour cela nous ajoutons une méthode d'accès `setVille` qui intègre une logique spécifique :

```
package

{

    public class Joueur
    {

        // définition des propriétés de la classe
        private var nom:String;
        private var prenom:String;
        private var age:Number;
        private var ville:String;

        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:Number,
pVille:String )
```

```
            {

                prenom = pPrenom;
                nom = pNom;
                age = pAge;
                ville = pVille;

            }

            // récupère le prénom du joueur
            public function getPrenom ( ):String

            {

                return prenom;

            }

            // récupère le nom du joueur
            public function getNom ( ):String

            {

                return nom;

            }

            // récupère l'age du joueur
            public function getAge ( ):Number

            {

                return age;

            }

            // récupère la ville du joueur
            public function getVille ( ):String

            {

                return ville;

            }

            // récupère la ville du joueur
            public function setVille ( pVille:String ):void

            {

                ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
    ).toLowerCase();

            }

            // méthode permettant au joueur de se presenter
            public function sePresenter ( ):void

            {

                trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
```

```
            }

            // affiche une description de l'objet Joueur
            public function toString ( ):String

            {

                 return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";

            }

        }

}
```

Le changement de ville est désormais contrôlé, si un nom de ville est passé, le formatage est automatique :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan" );

// affectation d'une ville ma formatée
monJoueur.setVille ( "ClEEVELAnd");

// affiche : Cleeveland
trace( monJoueur.getVille() );
```

La méthode `setVille` formate automatiquement la ville passée au format attendu. En définissant cette méthode, nous sommes assurés du formatage des villes associées à chaque joueur. Pratique n'est-ce pas ?

Afin d'éviter les erreurs d'exécution, nous pouvons affecter les propriétés d'un objet en exerçant un contrôle. Nous rendons notre objet intelligent.

Une autre situation pourrait être imaginée. Imaginons que l'application serveur gérant la connexion de nos joueurs ne puisse gérer des noms ou prénoms de plus de 30 caractères, nous intégrons ce contrôle au sein des méthodes `setNom` et `setPrenom.` Nous rajoutons au passage une méthode `setAge` arrondissant automatiquement l'âge passé :

```
package

{

    public class Joueur
    {

            // définition des propriétés de la classe
            private var nom:String;
            private var prenom:String;
            private var age:Number;
            private var ville:String;

            // fonction constructeur
```

```
		function Joueur ( pPrenom:String, pNom:String, pAge:Number,
	pVille:String )

		{

			prenom = pPrenom;
			nom = pNom;
			age = pAge;
			ville = pVille;

		}

		// récupère le prénom du joueur
		public function getPrenom ( ):String

		{

			return prenom;

		}

		// récupère le nom du joueur
		public function getNom ( ):String

		{

			return nom;

		}

		// récupère l'age du joueur
		public function getAge ( ):Number

		{

			return age;

		}

		// récupère la ville du joueur
		public function getVille ( ):String

		{

			return ville;

		}

		// permet de changer le prénom du joueur
		public function setPrenom ( pPrenom:String ):void

		{

			if ( pPrenom.length <= 30 ) prenom = pPrenom;

			else trace ("Le prénom spécifié est trop long");

		}

		// permet de changer le nom du joueur
		public function setNom ( pNom:String ):void
```

```
            {

                if ( pNom.length <= 30 ) nom = pNom;

                else trace ("Le nom spécifié est trop long");

            }

            // permet de changer l'age du joueur
            public function setAge ( pAge:Number ):void

            {

                // l'age passé est automatiquement arrondi
                age = Math.floor ( pAge );

            }

            // récupère la ville du joueur
            public function setVille ( pVille:String ):void

            {

                ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
).toLowerCase();

            }

            // méthode permettant au joueur de se présenter
            public function sePresenter ( ):void

            {

                trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

            }

            // affiche une description de l'objet Joueur
            public function toString ( ):String

            {

                return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";

            }

        }

}
```

Si un nom ou un prénom trop long est passé nous pouvons afficher un message avertissant le développeur tiers :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affectation du nouveau nom
// affiche : Le nom spécifié est trop long
monJoueur.setNom ( "UnNomTresTresTresTresTresTresLong" );

// affiche : Wonder
```

```
trace( monJoueur.getNom() );

// affectation d'un nouvel age
monJoueur.setAge ( 24.8 );

// affiche : 24
trace( monJoueur.getAge() );
```

En effectuant ce contrôle, nous sommes assurés que la propriété `nom` ne contiendra aucune chaine de caractères de plus de 30 caractères. Sans cette vérification faite au moment de l'affectation, nous serions obligés de tester la propriété `nom` avant de faire quoi que ce soit.

Les méthodes d'affectation et de récupération s'avèrent très pratiques, elles offrent la possibilité de délivrer un code sécurisé, portable et élégant. Pourtant certains développeurs préfèrent utiliser les méthodes en lecture/écriture qu'ils considèrent comme plus pratique.

L'utilisation d'une méthode pour récupérer ou affecter une propriété ne leur convient pas. ActionScript 3 propose une solution alternative, appelées méthodes en lecture/écriture.

Méthodes en lecture/écriture

Les méthodes de lecture et d'écriture plus communément appelées *getter/setter* sont la prolongation des méthodes d'accès. Leur intérêt reste le même, elles permettent d'encapsuler notre code en gérant l'affectation et la récupération de propriétés d'un objet mais à l'aide d'une syntaxe différente.

Les méthodes de lecture et d'écriture permettent au développeur d'appeler de manière transparente ces méthodes comme si ce dernier ciblait une propriété. Afin de bien comprendre ce concept nous allons intégrer des méthodes de lecture et d'écriture dans notre classe `Joueur`.

Afin de définir ces méthodes nous utilisons deux mots clés :

- `get` : Définit une méthode de lecture
- `set` : Définit une méthode d'écriture

Une méthode de lecture se définit de la manière suivante :

```
// affecte une propriété
public function get maPropriete ( ):type

{

    return proprieteInterne;

}
```

Celle-ci doit obligatoirement retourner une valeur, et ne posséder aucun paramètre au sein de sa signature. Si ce n'est pas le cas, une erreur à la compilation est générée.

Une méthode d'écriture se définit de la manière suivante :

```
// affecte une propriété
public function set maPropriete ( pNouvelleValeur:type ):void

{

    proprieteInterne = pNouvelleValeur;

}
```

Celle-ci doit obligatoirement posséder au moins un paramètre au sein de sa signature et ne retourner aucune valeur. Si ce n'est pas le cas, une erreur à la compilation est générée.

Nous allons modifier les méthodes de récupération et d'affectation `setNom` et `getNom` par des méthodes en lecture/écriture.

Afin d'éviter un conflit de définition de variables, nous devons nous assurer que les méthodes en lecture/écriture ne possèdent pas le même nom que les propriétés que nous souhaitons externaliser :

```
package

{

    public class Joueur
    {

         // définition des propriétés de la classe
         private var _nom:String;
         private var prenom:String;
         private var age:Number;
         private var ville:String;

         // fonction constructeur
         function Joueur ( pPrenom:String, pNom:String, pAge:Number,
pVille:String )

         {

              prenom = pPrenom;
              _nom = pNom;
              age = pAge;
              ville = pVille;

         }

         // récupère le prénom du joueur
         public function getPrenom ( ):String

         {

              return prenom;
```

```
}

// récupère l'age du joueur
public function getAge ( ):Number

{

     return age;

}

// récupère la ville du joueur
public function getVille ( ):String

{

     return ville;

}

// permet de changer le prénom du joueur
public function setPrenom ( pPrenom:String ):void

{

     if ( pPrenom.length <= 30 ) prenom = pPrenom;

     else trace ("Le prénom spécifié est trop long");

}

// récupère le nom du joueur
public function get nom ( ):String

{

     return _nom;

}

// permet de changer le nom du joueur
public function set nom ( pNom:String ):void

{

     if ( pNom.length <= 30 ) _nom = pNom;

     else trace ("Le nom spécifié est trop long");

}

// permet de changer l'age du joueur
public function setAge ( pAge:Number ):void

{

     // l'age passé est automatiquement arrondi
     age = Math.floor ( pAge );

}
```

```
            // récupère la ville du joueur
            public function setVille ( pVille:String ):void

            {

                 ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
).toLowerCase();

            }

            // méthode permettant au joueur de se présenter
            public function sePresenter ( ):void

            {

                 trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

            }

            // affiche une description de l'objet Joueur
            public function toString ( ):String

            {

                 return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";

            }

       }

}
```

En utilisant le mot clé `set` nous avons défini une méthode d'écriture. Ainsi, le développeur à l'impression d'affecter une propriété, en réalité notre méthode d'écriture est déclenchée :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affectation du nouveau nom
monJoueur.nom = "Womack";
```

Cela permet de conserver une affectation des données, dans la même écriture que les propriétés, en conservant le contrôle des données passées :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affectation du nouveau nom
// affiche : Le nom spécifié est trop long
monJoueur.nom = "UnNomTresTresTresTresTresTresLong";
```

En utilisant le mot clé `get` nous avons défini une méthode de lecture.

Ainsi, le développeur à l'impression de cibler une simple propriété, en réalité notre méthode de lecture est déclenchée :

```
// instanciation d'un joueur
```

```
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// affectation du nouveau nom
// affiche : Le nom spécifié est trop long
monJoueur.nom = "UnNomTresTresTresTresTresTresLong";

// récupération du nom
// affiche : Wonder
trace( monJoueur.nom ) ;
```

Voici le code final de notre classe `Joueur` :

```
package

{

    public class Joueur
    {

          // définition des propriétés de la classe
          private var _nom:String;
          private var _prenom:String;
          private var _age:Number;
          private var _ville:String;
          private var id:int;

          // fonction constructeur
          function Joueur ( pPrenom:String, pNom:String, pAge:Number,
pVille:String )

          {

               _prenom = pPrenom;
               _nom = pNom;
               _age = pAge;
               _ville = pVille;

          }

          // récupère le nom du joueur
          public function get nom ( ):String

          {

               return _nom;

          }

          // permet de changer le nom du joueur
          public function set nom ( pNom:String ):void

          {

               if ( pNom.length <= 30 ) _nom = pNom;

               else trace ("Le nom spécifié est trop long");

          }

          // récupère le prénom du joueur
          public function get prenom ( ):String
```

```
        {

            return _prenom;

        }

        // permet de changer le prénom du joueur
        public function set prenom ( pPrenom:String ):void

        {

            if ( pPrenom.length <= 30 ) _prenom = pPrenom;

            else trace ("Le prénom spécifié est trop long");

        }

        // permet de changer l'age du joueur
        public function set age ( pAge:Number ):void

        {

            // l'age passé est automatiquement arrondi
            _age = Math.floor ( pAge );

        }

        // récupère l'age du joueur
        public function get age ( ):Number

        {

            return _age;

        }

        // récupère la ville du joueur
        public function set ville ( pVille:String ):void

        {

            _ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
).toLowerCase();

        }

        // récupère la ville du joueur
        public function get ville ( ):String

        {

            return _ville;

        }

        // méthode permettant au joueur de se présenter
        public function sePresenter ( ):void

        {

            trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );
```

```
            }

            // affiche une description de l'objet Joueur
            public function toString ( ):String

            {

                 return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
" + age + ", ville : " + ville + "]";

            }

        }

}
```

L'encapsulation fait partie des points essentiels de la programmation orientée objet, les méthodes de récupération et d'affectation permettent de cacher l'implémentation et d'exercer un contrôle lors de l'affectation de propriétés. Les méthodes de lecture/écriture étendent ce concept en proposant une syntaxe alternative.

Libre à vous de choisir ce que vous préférez, de nombreux débats existent quand au choix d'utilisateur de méthodes de récupération et d'affectation et de lecture/écriture.

A retenir

- Dans la plupart des cas, pensez à rendre privées les propriétés d'un objet.
- Afin d'encapsuler notre code nous pouvons utiliser des méthodes de récupération ou d'affectation ou bien des méthodes de lecture et d'écriture.
- L'intérêt est d'exercer un contrôle sur la lecture et l'écriture de propriétés.
- Un code encapsulé rend la classe évolutive et solide.

Cas d'utilisation de l'attribut static

L'attribut de propriété `static` permet de rendre un membre utilisable dans un contexte de classe et non d'occurrence. En d'autres termes, lorsqu'une méthode ou une propriété est définie avec l'attribut `static` celle-ci ne peut être appelée que depuis le constructeur de la classe.

Les propriétés et méthodes statiques s'opposent aux méthodes et propriétés d'occurrences qui ne peuvent être appelées que sur des instances de classes. L'intérêt d'une méthode ou propriété statique est d'être globale à la classe, si nous définissons une propriété statique `nVitesse` au sein d'une classe `Vaisseau`, lors de sa modification tout les vaisseaux voient leur vitesse modifiée.

Parmi les méthodes statiques les plus connues nous pouvons citer :

- `Math.abs`
- `Math.round`
- `Math.floor`
- `ExternalInterface.call`

Nous utilisons très souvent ses propriétés statiques au sein de classes comme `Math` ou `System` :

- `Math.PI`
- `System.totalMemory`
- `Security.sandboxType`

Voici un exemple canonique d'utilisation d'une propriété statique. Afin de savoir combien d'instances de classes ont été crées, nous définissons une propriété statique `i` au sein de la classe `Joueur.`

A chaque instanciation nous incrémentons cette propriété et affectons la valeur à la propriété d'occurrence `id` :

```
package

{

    public class Joueur
    {

          // définition des propriétés de la classe
          private var nom:String;
          private var prenom:String;
          private var age:Number;
          private var ville:String;
          private var id:int;

          // propriété statique
          private static var i:int = 0;

          // propriété id
          private var id:int;

          // fonction constructeur
          function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

          {

               prenom = pPrenom;
               nom = pNom;
               age = pAge;
               ville = pVille;

               // à chaque objet Joueur crée, nous incrémentons
               // la propriété statique i
               id = i++;
```

```
            }

            // reste du code de la classe non montré

        }

    }
```

A chaque instanciation d'un objet `Joeur`, le constructeur est déclenché et incrémente la propriété statique `i` aussitôt affectée à la propriété d'occurrence `id`.

Pour pouvoir récupérer le nombre de joueurs crées, nous allons créer une méthode statique nous renvoyant la valeur de la propriété statique `i` :

```
// renvoie le nombre de joueurs créés
public static function getNombreJoueurs ( ):int

{

        return i;

}
```

Afin de cibler une propriété statique au sein d'une classe nous précisons par convention toujours le constructeur de la classe avant de cibler la propriété :

```
// fonction constructeur
function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String )

{

        prenom = pPrenom;
        nom = pNom;
        age = pAge;
        ville = pVille;

        id = Joueur.i++;

}

// renvoie le nombre de joueurs créés
public static function getNombreJoueurs ( ):int

{

        return Joueur.i;

    }
```

En spécifiant le constructeur avant la propriété, un développeur tiers serait tout de suite averti de la nature statique d'une propriété.

Voici le code final de la classe `Joueur` :

```
package

{

  public class Joueur

  {
        // fonction constructeur
        function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
)

        {

              prenom = pPrenom;
              nom = pNom;
              age = pAge;
              ville = pVille;

              id = Joueur.i++;

        }

        // renvoie le nombre de joueurs créés
        public static function getNombreJoueurs ( ):int

        {

              return Joueur.i;

        }

        // récupère le prénom du joueur
        public function getPrenom ( ):String

        {

              return prenom;

        }

        // récupère le nom du joueur
        public function getNom ( ):String

        {

              return nom;

        }

        // récupère l'age du joueur
        public function getAge ( ):int

        {

              return age;

        }

        // récupère la ville du joueur
        public function getVille ( ):String

        {
```

```
                return ville;

            }

            // récupère la ville du joueur
            public function setVille ( pVille:String ):void

            {

                ville = pVille.charAt(0).toUpperCase()+pVille.substr ( 1
    ).toLowerCase();

            }

            // permet de changer le nom du joueur
            public function setNom ( pNom:String ):void

            {

                if ( pNom.length <= 30 ) nom = pNom;

                else trace ("Le nom spécifié est trop long");

            }

            // permet de changer le prénom du joueur
            public function setPrenom ( pPrenom:String ):void

            {

                if ( pPrenom.length <= 30 ) prenom = pPrenom;

                else trace ("Le prénom spécifié est trop long");

            }

            // méthode permettant au joueur de se presenter
            public function sePresenter ( ):void

            {

                trace("Je m'appelle " + prenom + ", j'ai " + age + " ans." );

            }

            // affiche une description de l'objet Joueur
            public function toString ( ):String

            {

                return "[Joueur prenom : " + prenom +",  nom : " + nom + ", age :
    " + age + ", ville : " + ville + "]";

            }

      }

    }
```

Afin de savoir combien de joueurs ont été crées, nous appelons la méthode `getNombreJoueurs` directement depuis la classe `Joueur` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur("Stevie", "Wonder", 57, "Michigan");

// récupération du nom
// affiche : 1
trace( Joueur.getNombreJoueurs() );

// instanciation d'un joueur
var monSecondJoueur:Joueur = new Joueur("Bobby", "Womack", 57, "Detroit");

// récupération du nom
// affiche : 2
trace( Joueur.getNombreJoueurs() );
```

Si nous tentons d'appeler une méthode de classe sur une occurrence de classe :

```
// création d'un joueur connu :)
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");

// appel d'une méthode statique sur une instance de classe
monJoueur.getNombreJoueurs();
```

Le compilateur nous renvoie l'erreur suivante :

```
1061: Appel à la méthode getNombreJoueurs peut-être non définie, via la
référence de type static Joueur.
```

Contrairement aux propriétés d'occurrences de classes, les propriétés de classes peuvent être initialisées directement après leur définition. Ceci est du au fait qu'une méthode statique n'existe qu'au sein d'une classes et non des occurrences, ainsi si nous créons dix joueurs le constructeur de la classe `Joueur` sera déclenchée dix fois, mais l'initialisation de la classe et de ses propriétés statiques une seule fois. Ainsi notre tableau `tableauJoueurs` n'est pas recrée à chaque déclenchement du constructeur.

Il est impossible d'utiliser le mot clé `this` au sein d'une méthode de classe, seules les méthodes d'instances le permettent. Une méthode statique évolue dans le contexte d'une classe et non d'une instance. Si nous tentons de référencer `this` au sein d'une méthode statique, la compilation est impossible :

```
// renvoie le nombre de joueurs créés
public static function getNombreJoueurs ( ):int

{

  return this.i;

}
```

Le message d'erreur suivant s'affiche :

```
1042: Il est impossible d'utiliser le mot-clé this dans une méthode statique.
Il ne peut être utilisé que dans les méthodes d'une instance, la fermeture
d'une fonction et le code global.
```

A l'inverse les méthodes d'instances peuvent cibler une propriété ou méthode statique. Le constructeur de la classe `Joueur` incrémente la propriété statique `i` :

```
		// fonction constructeur
		function Joueur ( pPrenom:String, pNom:String, pAge:int, pVille:String
	)

		{

			prenom = pPrenom;
			nom = pNom;
			age = pAge;
			ville = pVille;

			// A chaque objet Joueur crée, nous incrémentons la propriété
	statique i
			id = Joueur.i++;

		}
```

Lorsque nous avons besoin de définir une propriété ou une méthode ayant une signification globale à toutes les instances de classes nous utilisons l'attribut de propriétés `static`. Certaines classes ne contiennent que des méthodes statiques, nous pourrions imaginer une classe `VerifFormulaire` disposant de différentes méthodes permettant de valider un formulaire.

Au lieu de créer une instance de classe puis d'appeler la méthode sur celle-ci, nous appelons directement la méthode `verifEmail` sur la classe :

```
// vérification du mail directement à l'aide d'une méthode statique
var mailValide:Boolean = OutilsFormulaire.verifEmail( "bobby@groove.com" );
```

Cette technique permet par exemple la création de classes utilitaires, rendant l'accès à des fonctionnalités sans instancier d'objet spécifique. Nous piochons directement sur la classe la fonctionnalité qui nous intéresse.

A retenir :

- L'utilisation du mot clé `this` est interdite au sein d'une méthode statique.
- A l'inverse, une méthode d'instance de classe peut cibler ou appeler une méthode ou propriété de classe.
- Les méthodes ou propriétés statiques ont un sens global à la classe.

La classe JoueurManager

La programmation objet prend tout son sens lorsque nous faisons travailler plusieurs objets ensemble, nous allons maintenant créer une

classe qui aura comme simple tâche d'ajouter ou supprimer nos joueurs au sein de l'application, d'autres fonctionnalités comme le tri ou autres pourront être ajoutées plus tard.

Le but de la classe `JoueurManager` sera de centraliser la gestion des joueurs de notre application. A coté de notre classe `Joueur` nous définissons une nouvelle classe appelée `JoueurManager` :

```
package

{

  public class JoueurManager

  {

        public function JoueurManager ( )

        {



        }

  }

}
```

Nous définissons une propriété d'occurrence `tableauJoueurs` afin de contenir chaque joueur :

```
package

{

  public class JoueurManager

  {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array = new Array();

        public function JoueurManager ( )

        {



        }

  }

}
```

La méthode `ajouteJoueur` ajoute la référence au joueur passé en paramètre au tableau interne `tableauJoueurs` et appelle la méthode `sePresenter` sur chaque joueur entrant :

```
package
```

```
{

  public class JoueurManager

  {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array = new Array();

        public function JoueurManager ( )

        {



        }

        public function ajouteJoueur ( pJoueur:Joueur ):void

        {

              pJoueur.sePresenter();

              tableauJoueurs.push ( pJoueur );

        }
  }

}
```

La méthode `ajouteJoueur` accepte un paramètre de type `Joueur`, seuls des objets de type `Joueur` pourront donc être passés. Lorsqu'un joueur est ajouté la méthode `sePresenter` est appelée sur celui-ci.

Ainsi, pour ajouter des joueurs nous créons chaque instance puis les ajoutons au gestionnaire de joueurs, la classe `JoueurManager` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");
var monDeuxiemeJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monTroisiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48, "Los
Angeles");

// gestion des joueurs
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs
// affiche :
/*
Je m'appelle Stevie, j'ai 57 ans.
Je m'appelle Bobby, j'ai 66 ans.
Je m'appelle Michael, j'ai 48 ans.
*/
monManager.ajouteJoueur ( monJoueur );
monManager.ajouteJoueur ( monDeuxiemeJoueur );
monManager.ajouteJoueur ( monTroisiemeJoueur );
```

Lorsqu'un joueur est ajouté, il se présente automatiquement. Au cas où un joueur serait supprimé par le système, nous devons le supprimer du gestionnaire, nous définissons une méthode `supprimeJoueur`.

Celle-ci recherche le joueur dans le tableau interne grâce à la méthode `indexOf` et le supprime si celui-ci est trouvé. Autrement, nous affichons un message indiquant que le joueur n'est plus présent dans le gestionnaire :

```
package

{

  public class JoueurManager

  {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array = new Array();

        public function JoueurManager ( )

        {



        }

        public function ajouteJoueur ( pJoueur:Joueur ):void

        {

              pJoueur.sePresenter();

              tableauJoueurs.push ( pJoueur );

        }

        public function supprimeJoueur ( pJoueur:Joueur ):void

        {

              var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

              if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

              else trace("Joueur non présent !");

        }

  }

}
```

Nous pouvons supprimer un joueur en passant sa référence. Afin d'externaliser le tableau contenant tout les joueurs, nous définissons une méthode d'accès `getJoueurs` :

```
package
{

  public class JoueurManager
  {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array;

        public function JoueurManager ( )

        {

              tableauJoueurs = new Array();

        }

        public function ajouteJoueur ( pJoueur:Joueur ):void

        {

              pJoueur.sePresenter();

              tableauJoueurs.push ( pJoueur );

        }

        public function supprimeJoueur ( pJoueur:Joueur ):void

        {

              var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

              if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

              else trace("Joueur non présent !");

        }

        public function getJoueurs ( ):Array

        {

              return tableauJoueurs;

        }

  }

}
```

Afin de récupérer l'ensemble des joueurs, nous appelons la méthode `getJoueurs` sur l'instance de classe `JoueurManager` :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");
var monDeuxiemeJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monTroisiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48, "Los
Angeles");

// gestion des joueurs
```

```
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs
monManager.ajouteJoueur ( monJoueur );
monManager.ajouteJoueur ( monDeuxiemeJoueur );
monManager.ajouteJoueur ( monTroisiemeJoueur );

// récupération de l'ensemble des joueurs
// affiche : [Joueur prenom : Stevie,  nom : Wonder, age : 57, ville :
Michigan],[Joueur prenom : Bobby,  nom : Womack, age : 66, ville :
Detroit],[Joueur prenom : Michael,  nom : Jackson, age : 48, ville : Los
Angeles]
trace( monManager.getJoueurs() );
```

Afin de supprimer des joueurs nous appelons la méthode `supprimeJoueur` en passant le joueur à supprimer en référence :

```
// instanciation d'un joueur
var monJoueur:Joueur = new Joueur ("Stevie", "Wonder", 57, "Michigan");
var monDeuxiemeJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monTroisiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48, "Los
Angeles");

// gestion des joueurs
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs
monManager.ajouteJoueur ( monJoueur );
monManager.ajouteJoueur ( monDeuxiemeJoueur );
monManager.ajouteJoueur ( monTroisiemeJoueur );

// suppression de deux joueurs
monManager.supprimeJoueur ( monJoueur );
monManager.supprimeJoueur ( monTroisiemeJoueur );

// récupération de l'ensemble des joueurs
// affiche : [Joueur prenom : Bobby,  nom : Womack, age : 66, ville :
Detroit]
trace( monManager.getJoueurs() );
```

La classe `JoueurManager` nous permet de gérer les différents joueurs crées, nous pourrions ajouter de nombreuses fonctionnalités. En programmation orientée objet il n'existe pas une seule et unique manière de concevoir chaque application. C'est justement la richesse du développement orienté objet, nous pouvons discuter des heures durant, de la manière dont nous avons développé une application, certains seront d'accord avec votre raisonnement d'autres ne le seront pas. L'intérêt est d'échanger idées et point de vue sur une conception.

Lorsque nous devons développer une application, nous sommes souvent confrontés à des problèmes qu'un autre développeur a surement rencontrés avant nous. Pour apporter des solutions concrètes à des problèmes récurrents nous pouvons utiliser des *modèles de conceptions*. Ces derniers définissent la manière dont nous devons séparer et concevoir notre application toujours dans un objectif d'optimisation, de portabilité et de réutilisation. Nous reviendrons bientôt sur les modèles de conception les plus utilisés.

Nos classes `Joueur` et `JoueurManager` sont désormais utilisées dans nos applications, mais le client souhaite introduire la notion de modérateurs. En y réfléchissant quelques instants nous pouvons facilement admettre qu'un modérateur est une sorte de joueur, mais disposant de droits spécifiques comme le fait d'exclure un autre joueur ou de stopper une partie en cours. Un administrateur possède donc toutes les capacités d'un joueur. Il serait redondant dans ce cas précis, de redéfinir toutes les méthodes et propriétés déjà définies au sein de la classe `Joueur` au sein de la classe `Administrateur`, pour réutiliser notre code nous allons utiliser l'héritage.

L'héritage

La notion d'héritage est un concept clé de la programmation orientée objet tiré directement du monde qui nous entoure. Tout élément du monde réel hérite d'un autre en le spécialisant, ainsi en tant qu'être humain nous héritons de toutes les caractéristiques d'un mammifère, au même titre qu'une pomme hérite des caractéristiques du fruit.

L'héritage est utilisé lorsqu'une relation de type « est-un » est possible. Nous pouvons considérer bien qu'un administrateur *est un* type de joueur et possède toutes ses capacités, et d'autres qui font de lui un modérateur. Lorsque cette relation n'est pas vérifiée, l'héritage ne doit pas être considéré, dans ce cas nous préférerons généralement la *composition*. Nous traiterons au cours du chapitre 10 intitulé *Héritage versus composition* les différences entre les deux approches, nous verrons que l'héritage peut s'avérer rigide et difficile à maintenir dans certaines situations.

Dans un contexte d'héritage, des classes filles héritent automatiquement des fonctionnalités des classes mères. Dans notre cas, nous allons créer une classe `Administrateur` héritant de toutes les fonctionnalités de la classe `Joueur`. La classe mère est généralement appelée *super-classe*, tandis que la classe fille est appelée *sous-classe*.

Nous allons définir une classe `Administrateur` à coté de la classe `Joueur`, afin de traduire l'héritage nous utilisons le mot clé `extends` :

```
package

{

  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur

  {
```

```
        public function Administrateur ( )

        {

        }

  }

}
```

Puis nous instancions un objet administrateur :

```
var monModo:Administrateur = new Administrateur();
```

A la compilation, l'erreur suivante est générée :

```
1203: Aucun constructeur par défaut n'a été défini dans la classe de base
Joueur.
```

Lorsqu'une classe fille étend une classe mère, nous devons obligatoirement passer au constructeur parent, les paramètres nécessaires à l'initialisation de la classe mère, pour déclencher le constructeur parent nous utilisons le mot clé `super` :

```
package

{

  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur

  {

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )

        {

              super ( pPrenom, pNom, pAge, pVille );

        }

  }

}
```

L'intérêt de l'héritage réside dans la réutilisation du code. La figure 8-5 illustre le concept d'héritage de classes :

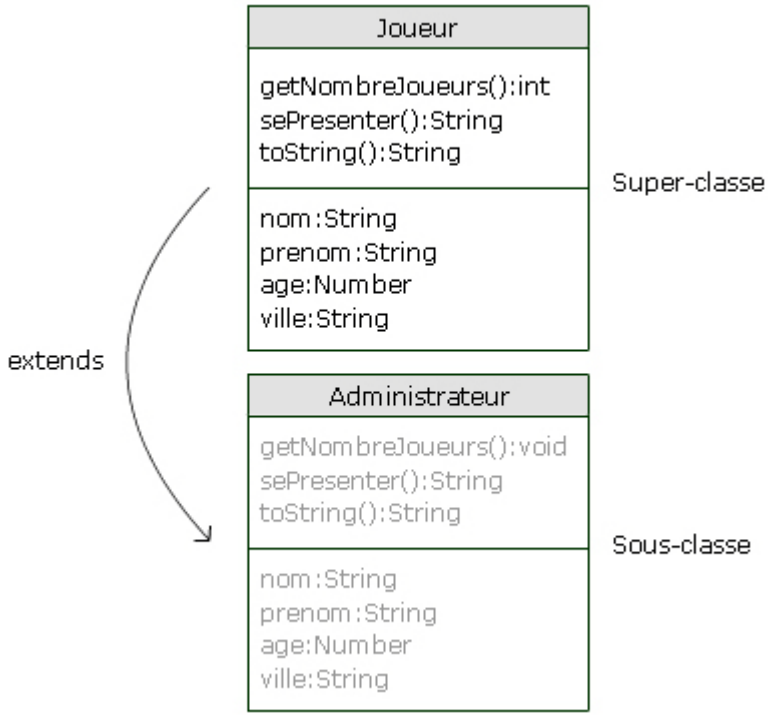

Figure 8-5. Héritage de classes.

Sans l'héritage nous aurions du redéfinir l'ensemble des méthodes de la classe `Joueur` au sein de la classe `Administrateur`.

Grâce à l'héritage exprimé par le mot clé `extends`, la classe fille `Administrateur` hérite de toutes les fonctionnalités de la classe mère `Joueur` :

```
// nous créons un objet administrateur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// le modérateur possède toutes les capacités d'un joueur
// affiche : Je m'appelle Michael, j'ai 48 ans.
monModo.sePresenter();

// affiche : Jackson
trace( monModo.nom );

// affiche : Michael
trace( monModo.prenom );

// affiche : 48
trace( monModo.age );

// affiche : Los Angeles
trace( monModo.ville );
```

Il existe quelques exceptions liées à l'héritage, les méthodes et propriétés statiques ne sont pas héritées. La méthode statique

`getNombreJoueurs` définie au sein de la classe `Joueur` n'est pas disponible sur la classe `Administrateur` :

```
// instanciation d'un administrateur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// la méthode statique Joueur.getNombreJoueurs n'est pas héritée
monModo.getNombreJoueurs();
```

L'erreur à la compilation suivante est générée :

```
1061: Appel à la méthode getNombreJoueurs peut-être non définie, via la
référence de type static Administrateur.
```

Notons que l'héritage multiple n'est pas géré en ActionScript 3, une classe ne peut hériter de plusieurs classes directes.

L'héritage ne se limite pas aux classes personnalisées, nous verrons au cours du chapitre 9 intitulé *Etendre Flash* comment étendre des classes natives de Flash afin d'augmenter les capacités de classes telles `Array`, `BitmapData` ou `MovieClip` et bien d'autres.

A retenir :

- L'héritage permet de réutiliser facilement les fonctionnalités d'une classe existante.
- La classe mère est appelée super-classe, la classe fille est appelée sous-classe. On parle alors de super-type et de sous-type.
- Afin d'hériter d'une classe nous utilisons le mot clé `extends`.

Sous-types et super-type

Grâce à l'héritage, la classe `Administrateur` possède désormais deux types, `Joueur` considéré comme le super-type et `Administrateur` comme sous-type :

```
// la classe Administrateur est aussi de type Joueur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// un modérateur est un joueur
// affiche : true
trace( monModo is Joueur );

// un modérateur est aussi un administrateur
// affiche : true
trace( monModo is Administrateur );
```

Partout où le super-type est attendu nous pouvons passer une instance de sous-type, ainsi une variable de type `Joueur` peut stocker un objet de type `Administrateur` :

```
// la classe Administrateur est aussi de type Joueur
```

```
var monModo:Joueur = new Administrateur("Michael", "Jackson", 48, "Los
Angeles");

// un modérateur est un joueur
// affiche : true
trace( monModo is Joueur );

// un modérateur est aussi un administrateur
// affiche : true
trace( monModo is Administrateur );
```

Tous les administrateurs sont des joueurs, le compilateur sait que toutes les fonctionnalités du type `Joueur` sont présentes au sein de la classe `Administrateur` et nous permet de compiler.

L'inverse n'est pas vrai, les joueurs ne sont pas forcément des administrateurs, il est donc impossible d'affecter un super-type à une variable de sous-type. Le code suivant ne peut être compilé :

```
// la classe Joueur n'est pas de type Administrateur
var monModo:Administrateur = new Joueur("Michael", "Jackson", 48, "Los
Angeles");
```

L'erreur à la compilation suivante est générée :

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

Le compilateur ne peut nous garantir que les fonctionnalités de la classe `Administrateur` seront présentes sur l'objet `Joueur` et donc interdit la compilation.

Nous retrouvons le même concept en utilisant les classes graphiques natives comme `flash.display.Sprite` et `flash.display.MovieClip`, le code suivant peut être compilé sans problème :

```
// une instance de MovieClip est aussi de type Sprite
var monClip:Sprite = new MovieClip();
```

La classe `MovieClip` hérite de la classe `Sprite` ainsi une instance de `MovieClip` est aussi de type `Sprite`. A l'inverse, un `Sprite` n'est pas de type `MovieClip` :

```
// une instance de Sprite n'est pas de type MovieClip
var monClip:MovieClip = new Sprite();
```

Si nous testons le code précédant, l'erreur à la compilation suivante est générée :

```
1118: Contrainte implicite d'une valeur du type statique flash.display:Sprite
vers un type peut-être sans rapport flash.display:MovieClip.
```

La question que nous pouvons nous poser est la suivante, quel est l'intérêt de stocker un objet correspondant à un sous-type au sein

d'une variable de super-type ? Pourquoi ne pas simplement utiliser le type correspondant ?

Afin de bien comprendre ce concept, imaginons qu'une méthode nous renvoie des objets de différents types, prenons un cas très simple comme la méthode `getChildAt` de la classe `DisplayObjectContainer`.

Si nous regardons sa signature, nous voyons que celle-ci renvoie un objet de type `DisplayObject` :

```
public function getChildAt(index:int):DisplayObject
```

En effet, la méthode `getChildAt` peut renvoyer toutes sortes d'objets graphiques tels `Shape`, `MovieClip`, `Sprite` ou bien `SimpleButton`. Tous ces types on quelque chose qui les lient, le type `DisplayObject` est le type commun à tous ces objets graphiques, ainsi lorsque plusieurs types sont attendus, nous utilisons un type commun aux différentes classes.

Nous allons justement mettre cela en application au sein de notre classe `JoueurManager`, si nous regardons la signature de la méthode `ajouterJoueur` nous voyons que celle-ci accepte un paramètre de type `Joueur` :

```
public function ajouteJoueur ( pJoueur:Joueur ):void

{

	pJoueur.sePresenter();

	tableauJoueurs.push ( pJoueur );

}
```

`Joueur` est le type communs aux deux classes `Joueur` et `Administrateur`, nous pouvons donc sans problème lui passer des instances de type `Administrateur` :

```
// création des joueurs et du modérateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var deuxiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48,
"Michigan");
var troisiemeJoueur:Joueur = new Joueur ("Lenny", "Williams", 50, "New
York");
var monModo:Administrateur = new Administrateur ("Stevie", "Wonder", 48, "Los
Angeles");

// gestion des joueurs
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs
/* affiche :
Je m'appelle Bobby, j'ai 66 ans.
Je m'appelle Michael, j'ai 48 ans.
```

```
Je m'appelle Bobby, j'ai 30 ans.
Je m'appelle Stevie, j'ai 48 ans.
Je suis modérateur
*/
monManager.ajouteJoueur ( premierJoueur );
monManager.ajouteJoueur ( deuxiemeJoueur );
monManager.ajouteJoueur ( troisiemeJoueur );
monManager.ajouteJoueur ( monModo );
```

Si nous devons plus tard créer de nouveaux types de joueurs en étendant la classe `Joueur`, aucune modification ne sera nécessaire au sein de la fonction `ajouteJoueur`.

A retenir :

- Grâce à l'héritage, une classe peut avoir plusieurs types.
- Partout où une classe mère est attendue nous pouvons utiliser une classe fille. C'est ce que nous appelons le *polymorphisme*.

Spécialiser une classe

Pour le moment, la classe `Administrateur` possède les mêmes fonctionnalités que la classe `Joueur`. Il est inutile d'hériter simplement d'une classe mère sans ajouter de nouvelles fonctionnalités, en définissant de nouvelles méthodes au sein de la classe `Administrateur` nous spécialisons la classe `Joueur`.

Nous allons définir une nouvelle méthode appelée `kickJoueur` qui aura pour but de supprimer un joueur de la partie en cours :

```
package

{

  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur

  {

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )

        {

              super ( pPrenom, pNom, pAge, pVille );

        }

        // méthode permettant de supprimer un joueur de la partie
        public function kickJoueur ( pJoueur:Joueur ):void

        {

              trace ("Kick " + pJoueur );

        }
```

```
    }
}
```

Désormais la classe `Administrateur` possède une méthode `kickJoueur` en plus des fonctionnalités de la classe `Joueur`. Nous avons spécialisé la classe `Joueur` à travers la classe `Administrateur`.

La figure 8-6 illustre les deux classes :

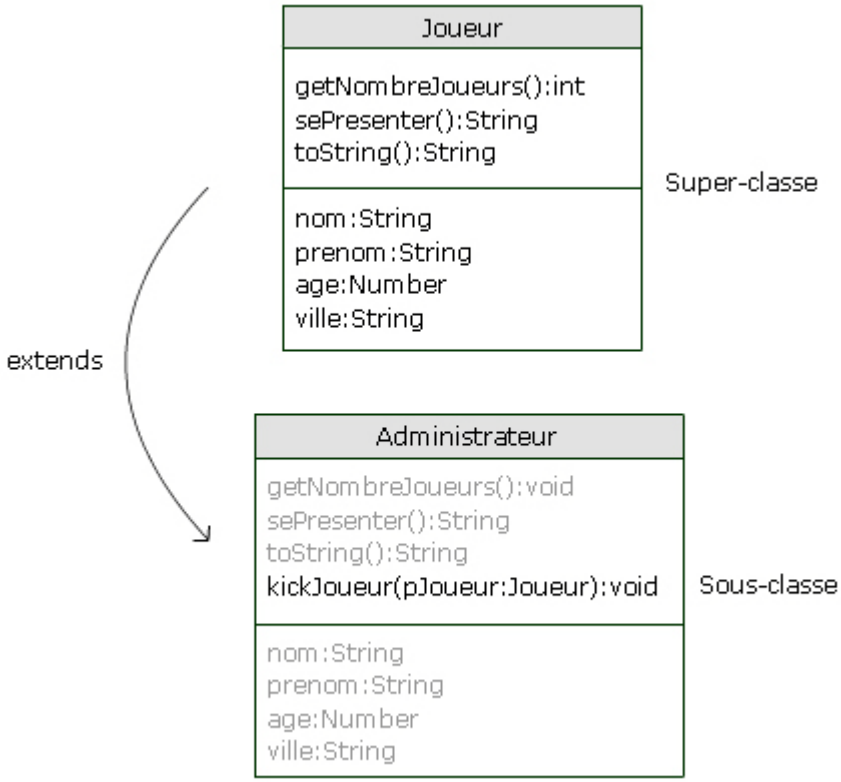

Figure 8-6. Spécialisation de la classe `Joueur`.

Lorsque la méthode `kickJoueur` est exécutée, nous devons appeler la méthode `supprimeJoueur` de la classe `JoueurManager` car c'est celle qui centralise et gère les joueurs connectés.

Chaque administrateur pourrait avoir une référence à la classe `JoueurManager` mais cela serait rigide, dans ce cas nous préférerons une approche événementielle en utilisant dans nos classes personnalisées le modèle événementiel d'ActionScript 3 à l'aide de la classe `flash.events.EventDispatcher`.

La notion de diffusion d'événements personnalisés est traitée en détail lors du chapitre 12 intitulé *Diffusion d'événements personnalisés*. Une

fois achevé, n'hésitez pas à revenir sur cet exemple pour ajouter le code nécessaire. L'idée est que la classe `Administrateur` diffuse un événement indiquant à la classe `JoueurManager` de supprimer le joueur en question.

Nous retrouverons ici le concept de diffuseur écouteur traité dans les chapitres précédents. La figure 8-6 illustre le concept :

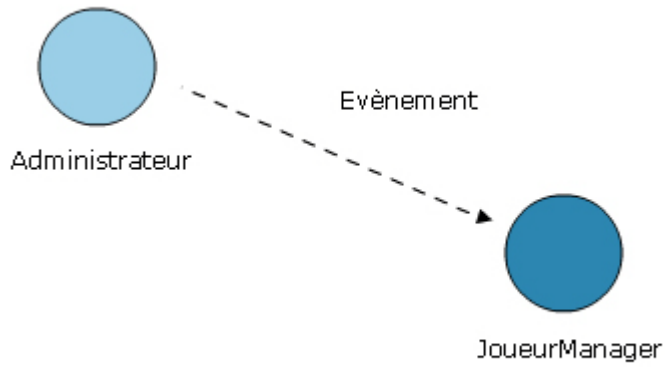

Figure 8-6. L'objet `JoueurManager` est souscrit à un événement auprès d'un administrateur.

Lorsqu'un administrateur diffuse l'événement approprié, l'objet `JoueurManager` supprime le joueur de la partie en cours. Nous allons nous intéresser maintenant à un processus appelé transtypage très utile dans un contexte d'héritage.

A retenir :

- Il ne faut pas confondre héritage et spécialisation.
- Même si cela est déconseillé, une classe peut hériter d'une autre sans ajouter de nouvelles fonctionnalités.
- Lorsqu'une sous-classe ajoute de nouvelles fonctionnalités, on dit que celle-ci spécialise la super-classe.

Le transtypage

Le transtypage est un processus très simple qui consiste à faire passer un objet pour un autre auprès du compilateur. Afin de comprendre le transtypage prenons la situation suivante, comme nous l'avons vu précédemment il peut arriver qu'une variable d'un type parent stocke des instances de types enfants.

Dans le code suivant nous stockons au sein d'une variable de type `Joueur` une instance de classe `Administrateur` :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monModo:Joueur = new Administrateur("Michael", "Jackson", 48, "Los
Angeles");

// la méthode kickJoueur n'existe pas pour le compilateur, car la classe
Joueur ne la définit pas
monModo.kickJoueur(premierJoueur);
```

Cela ne pose aucun problème, sauf lorsque nous tentons d'appeler la méthode `kickJoueur` sur la variable `monModo`, le compilateur nous génère l'erreur suivante :

```
1061: Appel à la méthode kickJoueur peut-être non définie, via la référence
de type static Joueur.
```

Pour le compilateur, la classe `Joueur` ne possède pas de méthode `kickJoueur` la compilation est donc impossible. Pourtant nous savons qu'à l'exécution la variable `monModo` contiendra une instance de la classe `Administrateur` qui possède bien la méthode `kickJoueur`. Afin de faire taire le compilateur nous allons utiliser le transtypage en disant en quelque sorte au compilateur « Fais moi confiance, il y'a bien la méthode `kickJoueur` sur l'objet, laisse moi compiler ».

La syntaxe du transtypage est simple, nous précisions le type puis nous plaçons deux parenthèses comme pour une fonction traditionnelle :

```
Type ( monObjet ) ;
```

Ainsi, nous transtypons la variable `monModo` vers le type `Administrateur` afin de pouvoir compiler :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var monModo:Joueur = new Administrateur("Michael", "Jackson", 48, "Los
Angeles");

// affiche : Kick [Joueur prenom : Bobby,  nom : Womack, age : 66, ville :
Detroit]
Administrateur ( monModo ).kickJoueur(premierJoueur);
```

A l'exécution, la méthode est bien exécutée. Là encore, nous pouvons nous demander dans quel cas concret nous devrions avoir recourt au transtypage ?

Nous allons définir une nouvelle méthode `joueurSon` sur la classe `Administrateur` qui sera automatiquement appelée au sein de la méthode `ajouteJoueur`. Lorsqu'un administrateur sera ajouté à

l'application un son sera joué afin de notifier de l'arrivée du modérateur :

```
package

{

  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur

  {

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )

        {

              super ( pPrenom, pNom, pAge, pVille );

        }

        // méthode permettant de supprimer un joueur de la partie
        public function kickJoueur ( pJoueur:Joueur ):void

        {

              trace ("Kick " + pJoueur );

        }

        // méthode permettant de jouer un son
        public function jouerSon ( ):void

        {

              trace("Joue un son");

        }

  }

}
```

Au sein de la classe `JoueurManager` nous devons appeler la méthode `jouerSon` lorsqu'un objet de type `Administrateur` est passé, pour cela nous testons si le type correspond puis nous appelons la méthode voulue :

```
package
{

  public class JoueurManager
  {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array;

        public function JoueurManager ( )

        {
```

```
            tableauJoueurs = new Array();

        }

        public function ajouteJoueur ( pJoueur:Joueur ):void

        {

            pJoueur.sePresenter();

            if ( pJoueur is Administrateur ) pJoueur.jouerSon();

            tableauJoueurs.push ( pJoueur );

        }

        public function supprimeJoueur ( pJoueur:Joueur ):void

        {

            var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

            if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

            else throw new Error ("Joueur non présent !");

        }

        public function getJoueurs ( ):Array

        {

            return tableauJoueurs;

        }

    }

}
```

Une fois nos classes modifiées, nous pouvons initialiser notre application :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var deuxiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48,
"Michigan");
var troisiemeJoueur:Joueur = new Joueur ("Lenny", "Williams", 50, "New
York");
var monModo:Administrateur  = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// gestion des joueurs
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs et administrateurs
monManager.ajouteJoueur ( premierJoueur );
monManager.ajouteJoueur ( deuxiemeJoueur );
monManager.ajouteJoueur ( troisiemeJoueur );
monManager.ajouteJoueur ( monModo );
```

A la compilation l'erreur suivante est générée :

```
1061: Appel à la méthode jouerSon peut-être non définie, via la référence de
type static Joueur.
```

Le compilateur se plaint car pour lui nous tentons d'appeler une méthode inexistante sur la classe `Joueur`. Nous savons qu'à l'exécution si la condition est validée, nous serons assurés que l'objet possède bien la méthode `jouerSon`, pour nous permettre de compiler nous transtypons vers le type `Administrateur`:

```
package
{

  public class JoueurManager
  {

        // tableau contenant les références de joueurs
        private var tableauJoueurs:Array;

        public function JoueurManager ( )

        {

              tableauJoueurs = new Array();

        }

        public function ajouteJoueur ( pJoueur:Joueur ):void

        {

              pJoueur.sePresenter();

              if ( pJoueur is Administrateur ) Administrateur ( pJoueur
).jouerSon();

              tableauJoueurs.push ( pJoueur );

        }

        public function supprimeJoueur ( pJoueur:Joueur ):void

        {

              var positionJoueur:int = tableauJoueurs.indexOf ( pJoueur );

              if ( positionJoueur != -1 ) tableauJoueurs.splice (
positionJoueur, 1 );

              else throw new Error ("Joueur non présent !");

        }

        public function getJoueurs ( ):Array

        {

              return tableauJoueurs;
```

```
        }

    }

}
```

Nous pouvons initialiser notre application :

```
// création d'un joueur et d'un administrateur
var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
var deuxiemeJoueur:Joueur = new Joueur ("Michael", "Jackson", 48,
"Michigan");
var troisiemeJoueur:Joueur = new Joueur ("Lenny", "Williams", 50, "New
York");
var monModo:Administrateur  = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// gestion des joueurs
var monManager:JoueurManager = new JoueurManager();

// ajout des joueurs et administrateurs
/* affiche :
Je m'appelle Bobby, j'ai 66 ans.
Je m'appelle Michael, j'ai 48 ans.
Je m'appelle Lenny, j'ai 50 ans.
Je m'appelle Michael, j'ai 48 ans.
Joue un son
*/
monManager.ajouteJoueur ( premierJoueur );
monManager.ajouteJoueur ( deuxiemeJoueur );
monManager.ajouteJoueur ( troisiemeJoueur );
monManager.ajouteJoueur ( monModo );
```

Nous reviendrons très souvent sur la notion de transtypage, couramment utilisée au sein la liste d'affichage.

A retenir :

- Il ne faut pas confondre conversion et transtypage.
- La transtypage ne convertit pas un objet en un autre, mais permet de faire passer un objet pour un autre.

Surcharge

Le concept de surcharge (*override* en anglais) intervient lorsque nous avons besoin de modifier certaines fonctionnalités héritées. Imaginons qu'une méthode héritée ne soit pas assez complète, nous pouvons surcharger celle-ci dans la sous-classe afin d'intégrer notre nouvelle version.

Nous allons définir au sein de notre classe `Administrateur` une méthode `sePresenter`, afin de surcharger la version héritée, pour cela nous utilisons le mot clé `override` :

```
package
```

```
{

  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur

  {

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )

        {

              super ( pPrenom, pNom, pAge, pVille );

        }

        // méthode permettant à l'administrateur de se présenter
        override public function sePresenter ( ):void

        {

              trace("Je suis modérateur");

        }

  }

}
```

En appelant la méthode `sePresenter` sur notre instance de modérateur nous déclenchons la version redéfinie :

```
// nous instancions un modérateur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// nous lui demandons de se présenter
// affiche : Je suis modérateur
monModo.sePresenter();
```

En ActionScript 2, il n'existait pas de mot clé pour surcharger une méthode, le simple fait de définir une méthode du même nom au sein de la sous-classe surchargeait la méthode héritée. Grâce au mot clé `override` introduit par ActionScript 3 il est beaucoup plus simple pour un développeur de savoir si une méthode est une méthode surchargeante ou non.

Attention, la méthode surchargeante doit avoir le même nom et la même signature que la méthode surchargée, sinon la surcharge est dite *non compatible*. Dans l'exemple suivant nous retournons une valeur lorsque la méthode `sePresenter` est exécutée :

```
package

{

  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur
```

```
  {

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )

        {

              super ( pPrenom, pNom, pAge, pVille );

        }

        // méthode permettant à l'administrateur de se présenter
        override public function sePresenter ( ):String

        {

              // déclenche la méthode surchargée
              super.sePresenter();

              return "Je suis modérateur";

        }

        // méthode permettant de supprimer un joueur de la partie
        public function kickJoueur ( pJoueur:Joueur ):void

        {

              trace ("Kick " + pJoueur );

        }

        // méthode permettant de jouer un son
        public function jouerSon ( ):void

        {

              trace("Joue un son");

        }

  }

}
```

La méthode surchargeante n'a pas la même signature que la méthode surchargée, la compilation est impossible, si nous testons le code suivant :

```
var monModo:Administrateur  = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");
```

L'erreur suivante est générée à la compilation :

```
1023: override non compatible.
```

Dans un contexte de surcharge, la méthode surchargée n'est pas perdue. Si nous souhaitons au sein de notre méthode surchargeante déclencher la méthode surchargée, nous utilisons le mot clé `super` :

```
package
```

```
{

  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur

  {

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )

        {

              super ( pPrenom, pNom, pAge, pVille );

        }

        // méthode permettant au joueur de se présenter
        override public function sePresenter ( ):void

        {

              // déclenche la méthode sePresenter surchargée
              super.sePresenter();

              trace("Je suis modérateur");

        }

  }

}
```

Ainsi, lorsque la méthode `sePresenter` est déclenchée, celle-ci déclenche aussi la version surchargée :

```
// nous instancions un modérateur
var monModo:Administrateur = new Administrateur("Michael", "Jackson", 48,
"Los Angeles");

// nous lui demandons de se présenter
// affiche :
/*
Je m'appelle Michael, j'ai 48 ans.
Je suis modérateur
*/
monModo.sePresenter();
```

Dans le code précédent nous augmentons les capacités de la méthode `sePresenter` en ajoutant le message « Je suis modérateur ». Grâce au mot clé `super` nous pouvons exécuter la méthode surchargée, celle-ci n'est pas perdue.

En surchargeant à l'aide d'une méthode vide, nous pouvons supprimer une fonctionnalité héritée :

```
package

{
```

```
  // l'héritage est traduit par le mot clé extends
  public class Administrateur extends Joueur

  {

        public function Administrateur ( pPrenom:String, pNom:String,
pAge:int, pVille:String )

        {

              super ( pPrenom, pNom, pAge, pVille );

        }

        // méthode surchargeante vide
        override public function sePresenter ( ):void

        {

        }

  }

}
```

Lorsque la méthode `sePresenter` est appelée sur une instance de la classe `Administrateur`, la version vide est exécutée. Celle-ci ne contenant aucune logique, nous avons annulé l'héritage de la méthode `sePresenter`.

A retenir :

- La surcharge permet de redéfinir une fonctionnalité héritée.
- Afin que la surcharge soit possible, la méthode surchargée et surchargeante doivent avoir la même signature.
- Afin de surcharger une méthode nous utilisons le mot clé *override*.