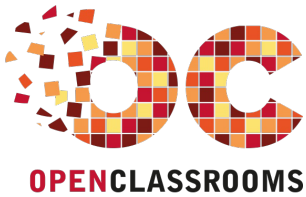
Apprenez à programmer en Actionscript 3

Par Akryum , Guillaume.



www.openclassrooms.com

Dernière mise à jour le 7/10/2013

Sommaire

Apprenez à programmer en Actionscript 3

Par  Akryum et  Guillaume.

Mise à jour : 07/10/2013
Difficulté : Intermédiaire Durée d'étude : 2 jours

Vous aimeriez apprendre à programmer en Actionscript 3.0 ?

Ce cours vous guidera pas à pas dans l'apprentissage de ce langage !

Bonjour à tous, amis Zéros !

Depuis plusieurs années, Flash s'est répandu sur le net et est maintenant quasi-omniprésent (pour le meilleur et pour le pire diront certains) : de nombreuses animations, jeux colorés, publicités, vidéos et musiques embellissent une majorité des sites web.
Une grande partie de ces animations est réalisée à l'aide de *Flash*, une technologie d'*Adobe Systems*.

Ce tutoriel vous propose de découvrir Flash, pour ensuite apprendre à programmer en Flash à l'aide de l'Actionscript 3 !

Au fait, que peut-on faire avec l'Actionscript ?

Depuis la version 3.0, l'Actionscript est devenu un langage de programmation à part entière, détaché des logiciels d'Adobe (notamment Flash Pro). Au départ, Flash a été conçu pour créer des animations vectorielles, aujourd'hui principalement utilisées dans la publicité en tant que bannières, ou pour embellir et dynamiser des sites web. Il est également possible de concevoir un **site 100% Flash** pour un maximum de dynamisme et d'interactivité. De plus, de récentes avancées dans le domaine de l'accélération matérielle par la carte graphique permettent de créer des jeux ou des applications en 3D complexe (l'Unreal Engine - Mass Effect, Borderlands 2, Batman, Gears of War... - a même été porté sur Flash Player) !
Avec l'arrivée des composants *Flex*, vous pouvez créer simplement des applications en ligne visuellement très avancées !
Enfin, avec *Adobe Air*, vous pouvez créer de véritables logiciels de bureaux en Actionscript 3 ou en HTML/CSS/Javascript ; ces applications sont également compatibles avec les principaux systèmes d'exploitation mobiles, comme **iOS**, **Android** ou **BlackBerry 10**, et les téléviseurs.

Avec un seul langage, vous pouvez réaliser toutes sortes d'applications et d'animations, pour le web, la bureautique, les mobiles, les téléviseurs...

Il ne vous reste plus qu'à vous lancer dans cette aventure !

Partie 1 : Les bases de l'Actionscript

Commençons par les bases du langage avec cette première partie : créer et compiler un projet, manipuler variables, conditions, boucles, fonctions et tableaux.

Vous avez dit « Actionscript » ?

Pour commencer ce cours en douceur, voici une petite introduction pour vous présenter la technologie Flash, ainsi que son langage de programmation principal, l'**Actionscript 3** !

Pour ceux qui ne connaîtraient pas ses possibilités, nous verrons ce qu'il est possible de faire en Flash, ainsi que les différentes utilisations de celui-ci. Nous essaierons également de présenter les technologies Flex et Air, qui peuvent servir à enrichir Flash. Quelques exemples de projets Flash vous seront donnés afin que vous puissiez vous faire une idée des possibilités liées à l'Actionscript.

Ce chapitre d'introduction ne présente aucune difficulté, même si vous êtes un parfait *débutant en programmation*. Je vous rappelle que ce cours est rédigé avec une difficulté progressive, aussi contentez-vous de lire celui-ci à votre rythme !

Adobe Flash

Présentation de Flash

Flash est une *technologie* actuellement développée et soutenue par Adobe Systems (prononcez "adobi").

Elle est principalement utilisée pour afficher des animations dynamiques et interactives dans des pages web, à travers le navigateur Internet. Elle permet par exemple d'ajouter une vidéo ou un jeu sur son site web, animer une galerie d'images, proposer une interface dynamique pour un service ou un logiciel en ligne (comme par exemple Photoshop Express).

Un document Flash est un fichier sous le format `swf` (**S**hock**w**ave **F**lash), et vous en avez sûrement ouvert plusieurs à chaque visite sur le web : en effet, ces *animations Flash*, couramment utilisées sur Internet, sont un assemblage d'images, de textes, de dessins et de sons pouvant s'animer et même interagir avec vous.

Parmi ses concurrents, on peut citer Silverlight de Microsoft ou encore Java de Sun/Oracle.
L'HTML5 couplé avec le CSS3 est une nouvelle alternative standardisée à ces animations, ne nécessitant pas de *plug-in* dans le navigateur. Les spécifications de l'HTML 5 ne sont malheureusement pas encore finalisées à l'heure actuelle. Je vous encourage à lire le tutoriel du Site du zéro sur l'HTML 5 et le CSS3 si vous vous sentez l'âme d'un webdesigner !

Un peu d'histoire

Flash n'est pas aussi récent que l'on pourrait le croire, car son ancêtre direct a vu le jour en 1995 et est sorti un an plus tard : il s'agit de **FutureSplash Animator**, un concurrent de Macromedia Shockwave à cette époque (un autre format multimédia orienté vers le web). FutureSplash a été développé par FutureWave Software, une compagnie à l'origine de SmartSketch, un logiciel de dessin vectoriel dont il manquait la composante animation, contrairement à Shockwave.

FutureSplash Animator

Son succès amena Macromedia à racheter FutureSplash Animator en décembre 1996, pour le renommer en *Macromedia Flash*, contraction de **F**uture et Spl**ash**.
En 2005, Adobe Systems acquiert Macromedia ; le développement de Flash continue pour aboutir à une évolution majeure de la technologie avec la sortie en 2007 de CS3 (*Creative Suite 3*) : c'est la naissance de l'**Actionscript 3** et de Flash 9.

Macromedia Flash

Depuis, Adobe travaille à l'amélioration du lecteur Flash, avec notamment la sortie en 2008 de la dixième version de Flash apportant quelques nouveautés et un début de support de l'accélération matérielle, pour obtenir de meilleures performances. En 2011, une onzième version majeure apporte une nouveauté très attendue : le support intégral de l'accélération matérielle par la carte graphique, ouvrant la voie à de nouveaux contenus en 3D complexe. Le fameux moteur de jeux vidéo d'Epic Games, l'Unreal Engine 3, a d'ailleurs été porté sur la plate-forme Flash !

Adobe cherche maintenant à faciliter l'utilisation de sa technologie sur les autres appareils multimédias, et on peut désormais développer en Flash sur un téléphone portable fonctionnant sous *Android* de Google, *iOs* d'Apple ou *BlackBerry 10* de RIM, sur des tablettes tactiles, et même sur des télévisions !

Les dérivés de Flash

La plupart des professionnels se servent du logiciel *Flash Professionnal* d'Adobe, mais il existe d'autres façons d'utiliser cette technologie. Voici les deux principales : la première utilise un autre langage tandis que la deuxième est en quelque sorte une extension de l'Actionscript 3.

Flex

Flex est un logiciel jumeau de *Flash Professionnal*, orienté développement et design d'applications. Il permet principalement de combler une lacune, si on peut dire, de Flash en termes de création d'interfaces utilisateur. C'est pourquoi celui-ci propose une grande collection de composants préconçus faciles à manipuler tels que des boutons, des champs de texte, etc. Les développeurs peuvent ainsi concevoir très rapidement une interface utilisateur pour des programmes qui ne nécessitent pas toute la puissance de dessin de Flash.

Adobe Flex

Flex utilise principalement un autre langage inventé par Macromédia : le MXML (**M**acromedia**Ex**tensible**M**arkup**L**anguage), une variante du langage très connu qu'est le XML. Le MXML est utilisé pour décrire la structure visuelle d'une application, de la même façon que pour écrire une page web avec l'HTML (lui aussi basé sur le XML) ; on place alors des balises représentant les composants du programme, tout en les mettant en forme.
Ensuite, l'Actionscript s'ajoute au XML dans une balise `<mx:Script>` et permet de manipuler et d'animer les balises MXML, ainsi qu'interagir avec l'utilisateur par exemple.

Pour obtenir plus d'informations, vous pouvez jeter un coup d'œil au tutoriel Flex de Migs.

Air

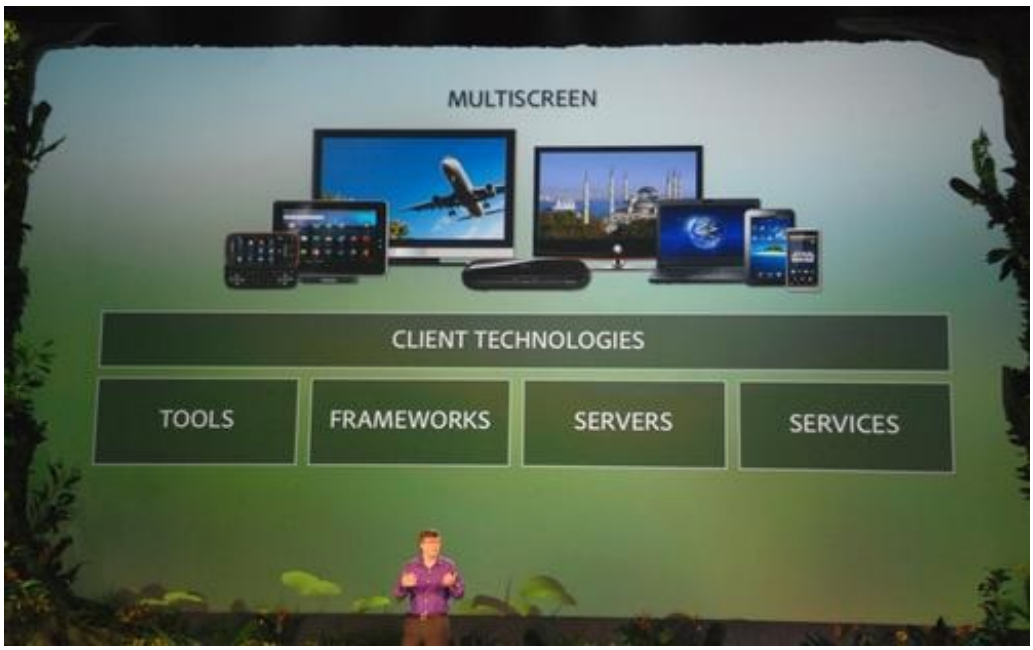
Adobe Air est une variante de Flash, permettant à n'importe quelle animation créée avec Flash ou Flex de s'installer et de fonctionner comme une véritable application. Grâce aux nombreuses nouvelles fonctionnalités et outils qui sont à votre disposition lorsque vous programmez une application Air, vous pouvez gérer le système de fichiers de l'ordinateur ou les disques amovibles, créer des bases de données, monter un serveur... Ainsi, peut-on programmer un traitement de texte ou encore un logiciel de messagerie instantanée, voire un navigateur Internet avec Flash.


Adobe Air

Une autre particularité de cette plate-forme ressemblant à Flash, est que, en plus du MXML et de l'Actionscript 3, on peut utiliser uniquement du simple HTML, CSS et Javascript pour créer de telles applications ; Air utilise alors le moteur Webkit (utilisé par Chrome et Safari entre autres) pour afficher ces éléments de pages web dans les applications. Ceci représente un avantage non négligeable pour les développeurs web voulant programmer des applications de bureau, sans toutefois avoir à apprendre et maîtriser un autre langage de programmation !

Enfin, c'est Air qui permet aux développeurs Flash de créer des applications à destination de mobiles comme l'iPhone, sur les

tablettes tactiles ou encore sur les téléviseurs, et cela depuis 2010 avec la sortie d'*Air 2.5*.



Présentation d'Adobe AIR 2.5, un dérivé de Flash.

Quelques exemples d'utilisation

Pour bien cerner les possibilités de Flash et bien différencier les différentes variantes de cette technologie, voici rien que pour vous quelques exemples illustrés.

Le dynamisme apporté au web par l'Actionscript

Il est probable que la première fois que vous ayez entendu parler de Flash soit en jouant à divers jeux sur le Web. En effet, il existe de nombreux sites web proposant des jeux flash. Bien entendu, ceux-ci sont réalisés en Flash à l'aide du langage Actionscript.
Pour vous donner un exemple concret, je vous ai sélectionné un jeu nommé Kingdom Rush dont un aperçu est donné ci-dessous :

Un jeu réalisé en Flash : Kingdom Rush.

Création de sites web orientés vers le visuel

La technologie Flash est aussi grandement utilisée dans la conception des sites web eux-mêmes. En général, les animations Flash sont plutôt réservées aux sites web statiques ou sites-vitrines. Ceux-ci sont très appréciés pour leurs interactions et leurs animations qui dynamisent énormément la navigation : il est important, notamment pour les sites commerciaux à destination du grand public, d'attirer le regard. Toutefois, il est tout à fait possible de créer des sites web dynamiques, et interagir avec un serveur grâce au langage PHP par exemple.
Pour que vous puissiez mieux vous faire une idée de la chose, je vous propose d'aller visiter le site de la nouvelle série de Canal+ : Carlos.

Le site de la série Carlos est réalisé en Flash.

Introduction d'Adobe Air

Comme nous le verrons dans le prochain chapitre, l'utilisation de la technologie Flash nécessite un lecteur spécifique nommé *Flash Player*. Celui-ci est très répandu à l'intérieur des navigateurs Internet, et l'utilisation de Flash s'est ainsi longtemps limitée au web. Heureusement ceci n'est plus le cas avec l'arrivée d'Adobe Air. En effet, cette technologie vient ajouter de nouvelles fonctionnalités à Flash et permettre l'utilisation de Flash hors-ligne en tant que programme.
Pour vous donner un exemple, l'application Flash du site de poker Winamax, disponible initialement sur navigateur Internet, a pu être convertie en vrai logiciel grâce à la technologie Adobe Air.

Le logiciel Winamax utilise la technologie Air.

Ainsi l'utilisation d'Adobe Air permet de porter les applications directement sur un système d'exploitation. Notamment cette technologie est actuellement utilisée par de nombreux systèmes d'exploitation mobiles tels que l'iOS, Android ou BlackBerry 10.

L'Actionscript 3

L'Actionscript est le langage de programmation servant à faire fonctionner les animations Flash ; c'est le plus utilisé parmi les langages de Flash. Il est basé sur l'ECMAscript, à l'instar du *Javascript* utilisé sur Internet, directement dans les pages web. Sans lui, aucune animation ou interaction dans un fichier Flash ne serait possible, et celui-ci serait alors réduit à une bête image fixe.

L'Actionscript est un langage **orienté objet**, de **haut niveau** et **événementiel**.

Orienté objet

En effet, sa structure est basée sur le concept d'**objet**, c'est-à-dire que tous les éléments de l'animation (y compris ceux qui ne relèvent pas de l'affichage, comme par exemple les nombres) sont des objets, avec des **attributs** et des **méthodes** qui leur sont attachés. Chaque objet est décrit par une **classe** : un ensemble d'attributs et de méthodes qui représentent son comportement.

Prenons comme exemple une voiture de sport : c'est un objet qui a pour classe `Voiture`, c'est-à-dire qu'elle a par exemple un attribut `vitesse` qui nous permet de connaître sa vitesse de déplacement, ou encore une méthode `tourner à gauche`, qui la fait tourner. Tous ces attributs et ces méthodes sont décrits et expliqués dans la classe `Voiture`.

S'il nous prenait l'envie de programmer un jeu de course de voitures basique, il faudrait d'abord écrire une classe `Voiture`, puis sur le fichier principal créer un nouvel objet de la classe `Voiture` pour créer une voiture (et ensuite la déplacer par exemple).

Une partie toute entière sera également consacrée à la programmation orientée objet, ne vous en faites pas si vous n'arrivez pas à tout bien saisir maintenant.

De haut niveau

L'Actionscript est un langage dit de **haut niveau**, c'est-à-dire que son fonctionnement est très éloigné du fonctionnement de l'ordinateur au niveau matériel, au contraire des langages dit de **bas niveau**, proches de la machine (comme le langage C).

Généralement, cela signifie qu'il est plus facile de réaliser certaines choses ou qu'il faut moins d'instructions, mais cela se traduit souvent par des performances plus faibles.

Le Java, le Python ou le Ruby sont d'autres exemples de langages de haut niveau.

Évènementiel

Enfin, c'est un langage évènementiel, c'est-à-dire que l'interactivité de vos programmes sera basée sur des **événements** que nous allons **écouter**. Par exemple, pour utiliser un bouton, nous allons écouter (donc attendre) sur lui l'événement « cliqué » qui sera déclenché lorsque l'utilisateur appuiera sur ce bouton. Bien sûr, pendant ce temps, nous pouvons faire autre chose : c'est tout l'intérêt de ce système.

En résumé

- L'Actionscript est un langage interprété, il a donc besoin d'un interpréteur (le lecteur Flash) pour être exécuté.
- La technologie Flash regroupe les programmes Flash classiques, les applications Flex que l'on peut trouver sur des pages web, ainsi que les applications AIR à destination du bureau ou d'autres appareils comme les mobiles.
- Le principal langage utilisé dans ces programmes est l'Actionscript 3, mais on peut aussi utiliser le MXML (pour Flex et AIR), voire l'HTML et le Javascript (pour les applications AIR uniquement).
- L'Actionscript 3 est un langage de **haut-niveau** : son fonctionnement est éloigné de la machine et il est en général plus facile d'implémenter des fonctionnalités complexes.
- C'est aussi un langage **orienté objet** : son fonctionnement est basé sur des concepts d'**objets** et de **classes**.
- Enfin, c'est un langage **événementiel** : on écoute des objets, et si des événements surviennent, on peut exécuter du code.

Votre premier programme avec Flex SDK

Maintenant que vous en savez un peu plus sur Flash, il est grand temps de passer à la pratique et l'essentiel de ce cours : *l'Actionscript 3* !
Vous aurez besoin d'outils pour suivre ce cours et appliquer ce que vous apprendrez (c'est très important), mais ne vous inquiétez pas, tous sont *gratuits*.

Le compilateur, qui servira à transformer votre code en animation est effectivement gratuit pour tout le monde, et il existe même une version *open-source* !
Il s'agit de **Flex SDK**, qui fait parti du projet *Adobe Open Source*.

Nous utiliserons également un logiciel pour nous aider à coder et à compiler nos projets, disponible hélas que sous Windows pour l'instant.

Cette partie est un peu difficile, surtout si vous n'avez jamais programmé : accrochez-vous, relisez les parties que vous n'avez pas comprises ! La suite n'en sera que d'autant plus facile à appréhender.
La partie sur la compilation en ligne de commande est facultative si vous travaillez sur Windows et que vous êtes allergiques à la console...

Préambule

Le compilateur

Un compilateur est un programme très utile dans la plupart des langages de programmation.
En réalité, vous vous doutez bien que l'ordinateur ne sait pas interpréter directement le code que nous lui écrivons : en effet, les langages de programmation ont été conçus pour être facilement utilisables par les êtres humains comme vous et moi. Or les ordinateurs ne comprennent que les instructions en binaire de bas niveau (cf. chapitre précédent). Il faut donc traduire nos programmes grâce au compilateur !

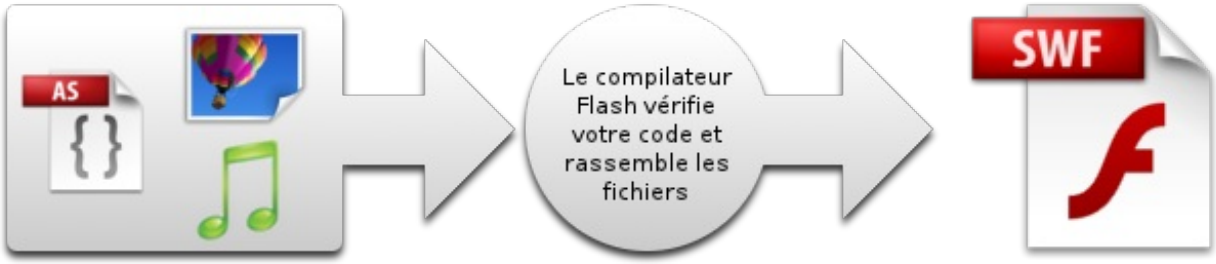
Le principe de la compilation

Certains langages de programmation sont interprétés par un logiciel faisant l'intermédiaire entre le code et l'ordinateur : on peut en général se passer de la compilation. C'est le cas de la plupart des langages utilisés sur Internet, comme le Javascript ou le PHP. *Mais c'est aussi le cas de l'Actionscript !*

Mais alors, pourquoi doit-on compiler nos programmes écrits en Actionscript ?

En Actionscript 3, la compilation vous donne des informations sur vos erreurs de syntaxe pour que vous les corrigiez plus facilement, mais elle permet surtout de rassembler tout votre code et le contenu nécessaire au bon déroulement de votre programme (comme les librairies) dans un seul fichier. Ainsi, il est plus facile d'intégrer une animation Flash dans un site web, et il devient possible d'importer directement dans l'animation des images, des sons, des polices de caractères ou d'autres médias qui seront chargés en même temps que votre programme. En outre, le compilateur compresse votre animation afin qu'elle prenne moins de temps à se charger.

Ainsi, en Flash, le compilateur ne traduit pas votre code en binaire. À vrai dire, ce n'est qu'un demi-compilateur : il ne s'occupe que de lier plusieurs fichiers en un seul (opération appelée *Édition de liens*), alors que les compilateurs classiques traduisent également le code.



Le compilateur Flash

Le fichier qui est produit est en général au format swf (*Shock Wave Flash*) ; c'est lui qui sera chargé par un navigateur Internet par exemple.

Le lecteur Flash

Donc il nous faut un logiciel pour que l'ordinateur comprenne notre code ?

Oui ! C'est là que le lecteur Flash (ou **Flash Player** en anglais) entre en scène. Ce logiciel contient un ensemble d'outils permettant d'exécuter votre programme : Flash Player est un interpréteur, et l'Actionscript est un langage interprété.

Le lecteur Flash

Tout ce processus se déroule à l'intérieur d'une machine virtuelle. Concrètement, Flash Player cache le système sur lequel votre programme tourne et gère lui-même la mémoire et les autres échanges nécessaires au bon déroulement de votre programme. Ainsi, ce dernier peut fonctionner correctement sur plusieurs systèmes d'exploitation (Windows, Mac, Linux, Android...) totalement différents.
Entre autres, il dispose d'un compilateur à la volée, ou *JIT* (*Just In Time*) qui traduit en temps réel certains passages de votre code en binaire afin d'améliorer les performances.

Installation des outils

Ainsi, pour programmer en Actionscript, il nous faut plusieurs outils, dont un compilateur, et un lecteur Flash. Nous allons dès à présent nous occuper de tout cela !

Java

Java est une technologie similaire à Flash, développée initialement par *Sun Microsystems* (racheté par *Oracle*), qui regroupe un ensemble d'outils permettant de créer des applications.
Or, le compilateur de Flash est écrit en Java : il faut donc que Java soit installé sur votre ordinateur.
Si vous êtes certains que c'est déjà le cas, vous pouvez passer à l'étape suivante.
Sinon, téléchargez et installez Java en suivant ce lien.

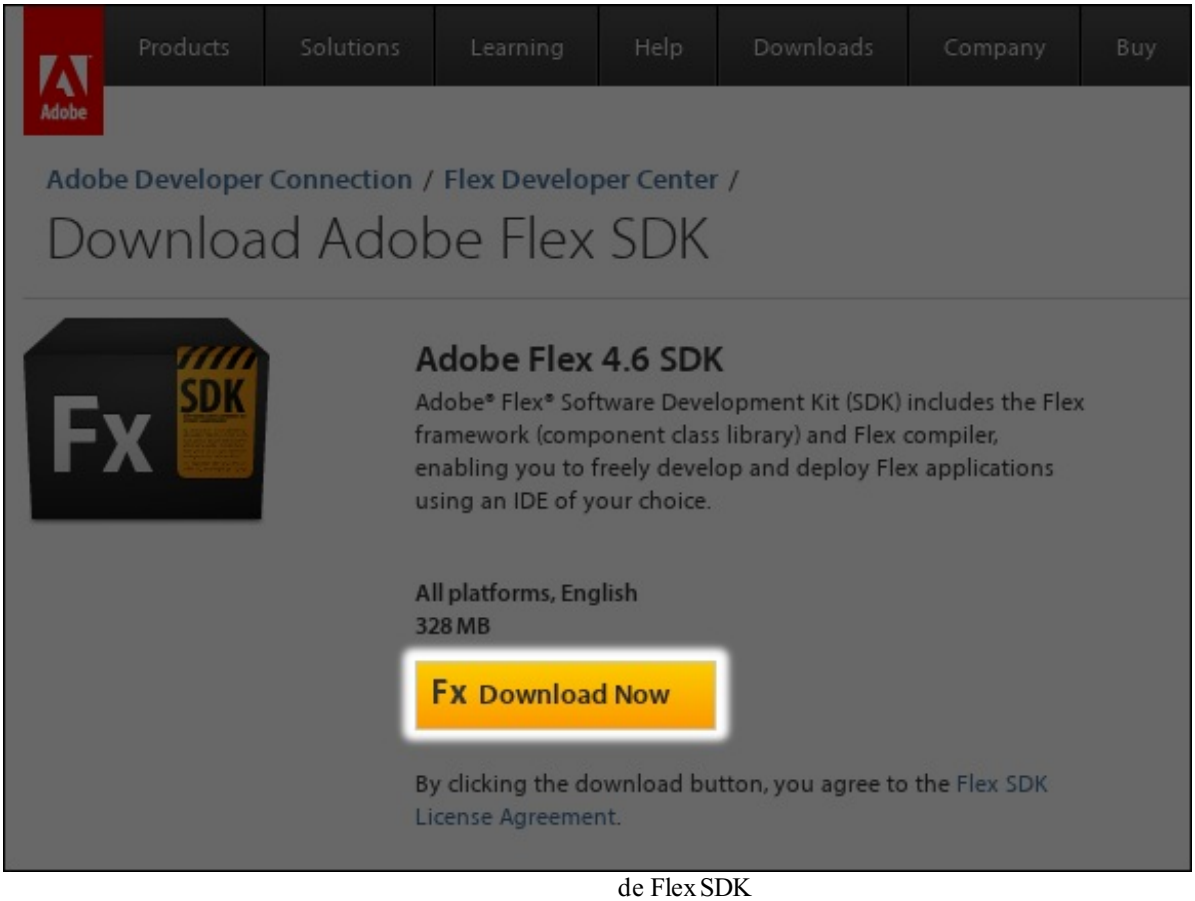
Le logo Java

Flex SDK

Le Flex SDK (*Software Development Kit*) est un ensemble d'outils de développement qui permettent entre autres de compiler des programmes écrits en Actionscript 1, 2 ou 3, et des applications Flex ou AIR.

Nous verrons dans la dernière partie de ce chapitre, comment utiliser un logiciel nous permettant de développer en Actionscript sans utiliser la console de lignes de commande : il s'agit de Flashdevelop (Windows uniquement). Si vous ne voulez pas essayer de compiler avec la console et que vous êtes sous Windows, ou si vous avez des problèmes de compilation, vous pouvez vous contenter de lire le tutoriel jusqu'à cette section.

Tout d'abord, il faut que vous récupériez le SDK sur le site officiel d'Adobe : télécharger Flex SDK.
Une fois sur le site d'Adobe, cliquez sur le bouton `Fx Download Now`, lancez le téléchargement et allez prendre un café  :



Téléchargement de Flex SDK

Une fois le téléchargement terminé, extrayez l'archive dans un répertoire facilement accessible (votre dossier personnel par exemple) et renommez-le pour simplifier son nom (par exemple, `Flex SDK 4`).

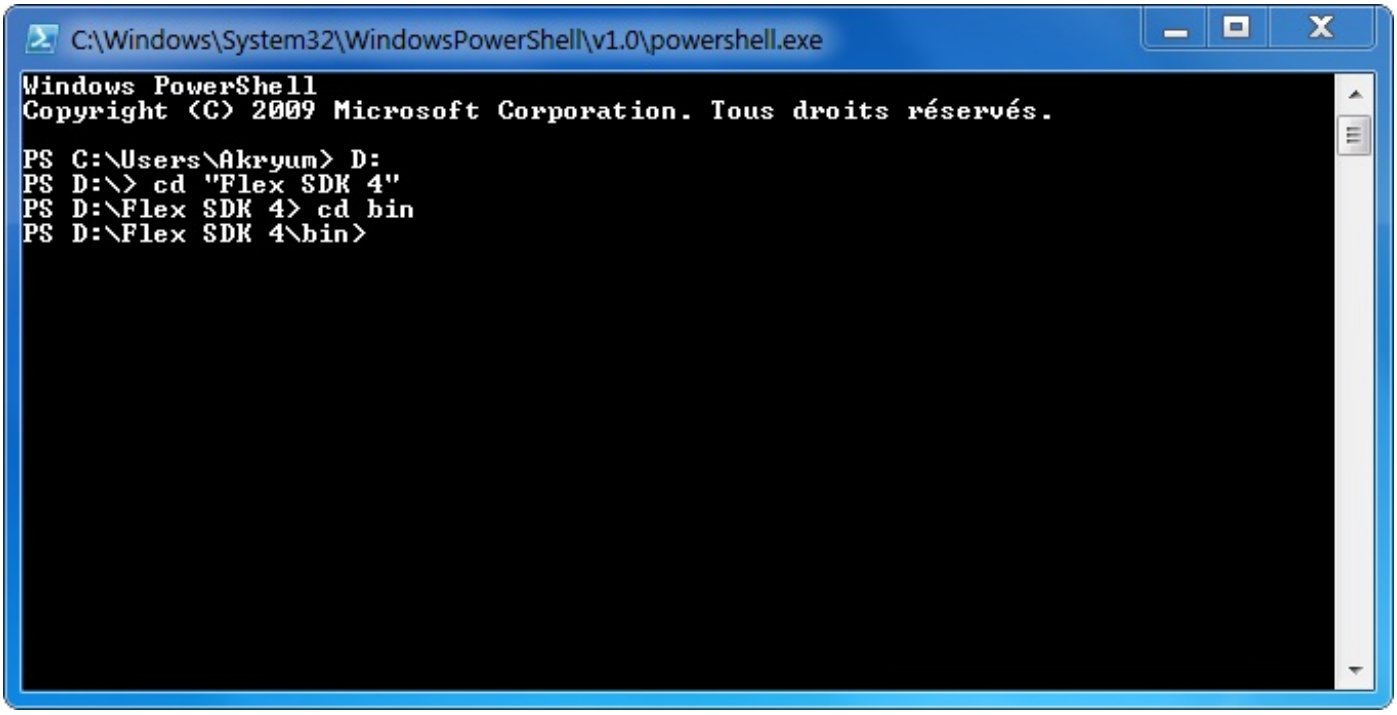
Vous voilà armés d'un compilateur Flash gratuit ! Mais comment s'en sert-on ? Avec les lignes de commande pardi ! 
Pour les utilisateurs de Windows, il est grand temps de renouer une relation avec le terminal ! Pour Mac et Linux en revanche, cela ne devrait pas trop vous poser de problèmes...

Voici la marche à suivre pour lancer un terminal :

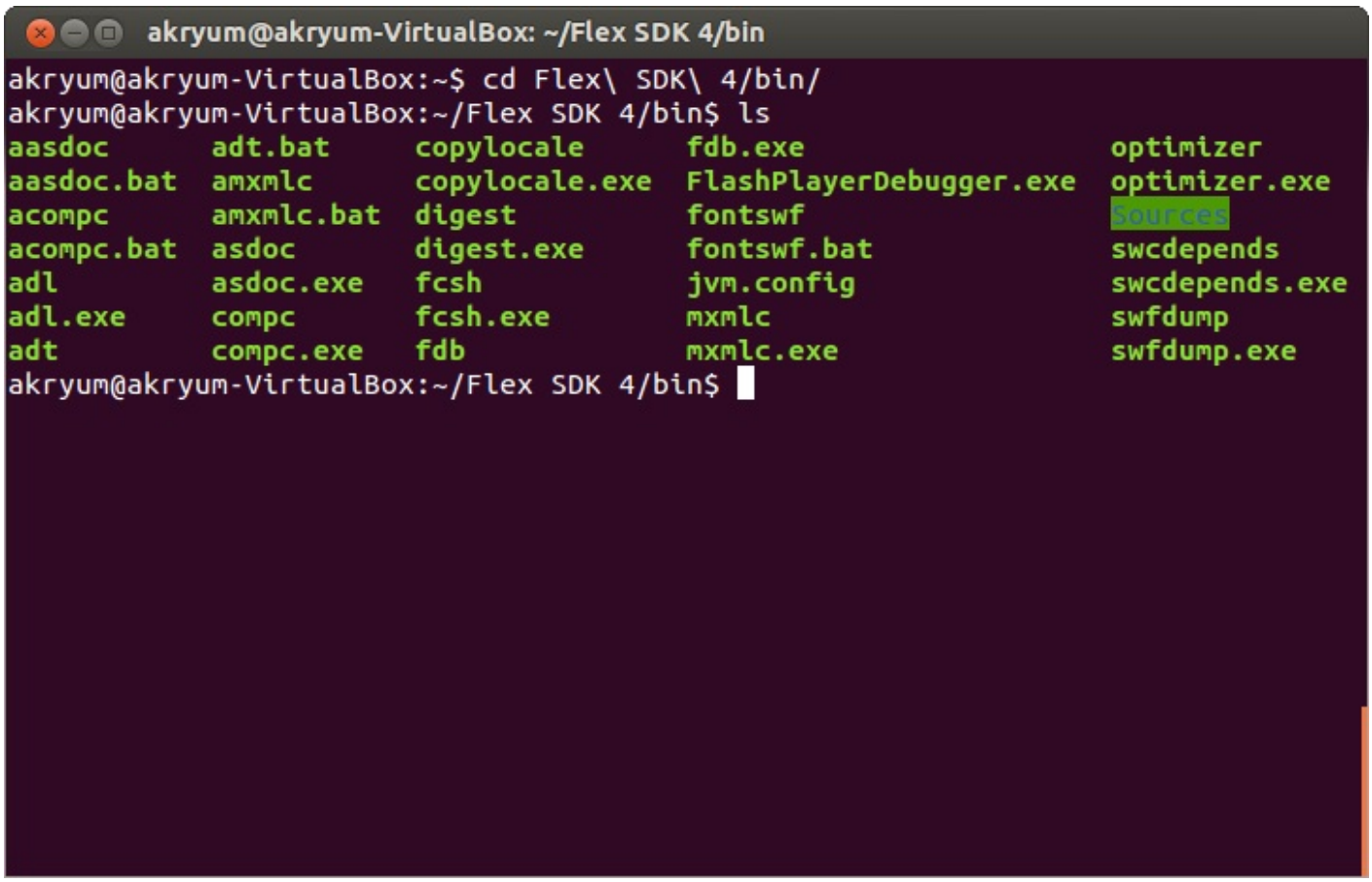
- **Windows :** dans le menu démarrer, allez dans `Tous les programmes`, `Accessoires`, `Invite de commandes`, ou appuyez sur Windows+R et entrez `cmd` puis validez.
- **Linux :** comment ça, vous ne savez pas ouvrir un terminal ? Le *gnome-terminal (Gnome)* ou la *Konsole (KDE)* conviendront parfaitement.
- **Mac :** dans le `Finder`, sélectionnez `Applications`, puis `Utilitaires` et enfin `Terminal`.

Sur Windows, il existe une alternative au terminal classique, bien plus complète et confortable, car elle se rapproche de ceux des systèmes d'exploitation Linux ou Mac. Il s'agit de *Powershell* ; pour le lancer, appuyez sur Windows+R, entrez `powershell` et validez. Désormais, vous pouvez par exemple appuyer sur TAB pour compléter automatiquement une commande ou un chemin, comme sur Linux !

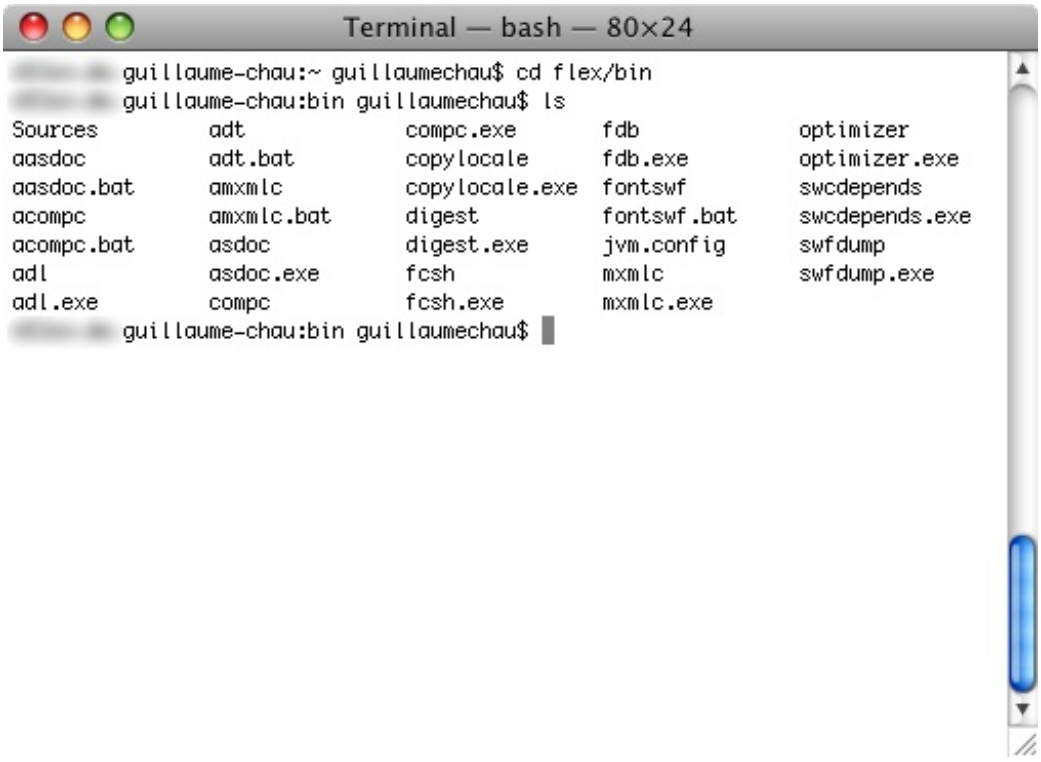
Maintenant, il va falloir se placer dans le dossier `bin` du SDK : utilisez la commande `cd` (Change Directory) pour vous déplacer dans l'arborescence de fichiers.



Windows



Linux



Mac

Version de débogage du lecteur Flash

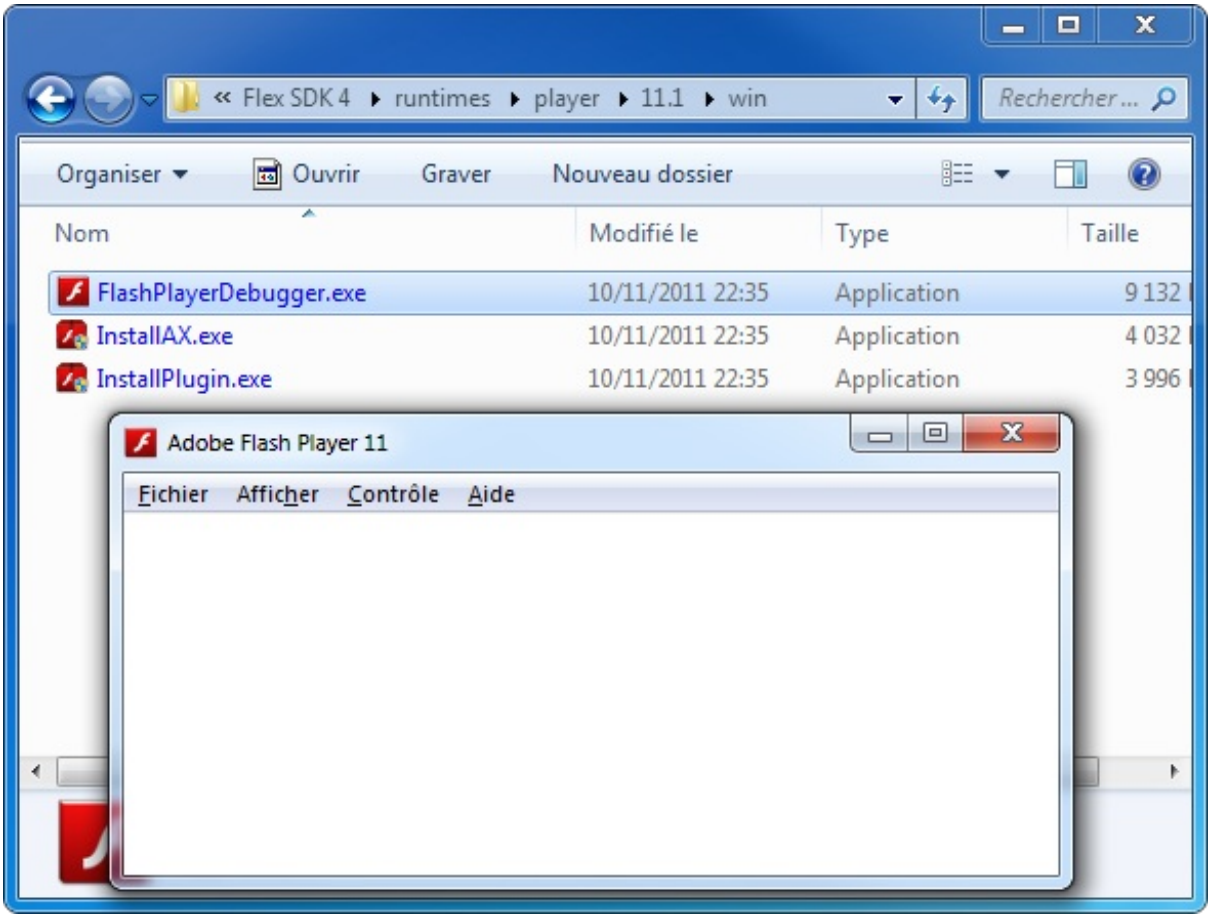
Un deuxième outil va nous être indispensable : il s'agit d'une version légèrement différente du lecteur Flash habituel que l'on trouve dans les navigateurs Internet par exemple. En effet, ce Flash Player de débogage peut s'exécuter tout seul comme une quelconque application, et dispose surtout de fonctions de débogage très utiles. Par exemple, si un problème survient pendant que l'on teste notre programme, une fenêtre nous affichera les détails de l'erreur, la ligne incriminée et les différents appels de

fonctions ayant conduit à cette erreur.

Vous pouvez dès à présent récupérer le lecteur. Il se trouve normalement dans le dossier `Flex SDK 4/runtimes/player/11.1` (la version peut varier). Choisissez le lecteur qui convient à votre système, appelé `Flash Player Debugger` (dans le dossier `lnx` pour Linux, `mac` pour Mac et `win` pour Windows).

Sur Windows

Lancez le fichier `FlashPlayerDebugger.exe` une fois afin que les fichiers `.swf` soient automatiquement associés au Lecteur Flash :



FlashPlayerDebugger.exe

Sur Linux

Commencez par extraire l'archive `flashplayerdebugger.tar.gz`.
Pour que son utilisation soit plus facile, et surtout parce que l'on en aura besoin un peu plus loin, vous pouvez déplacer l'exécutable `flashplayerdebugger` vers le dossier `/usr/bin` et le renommer en `flashplayer` :

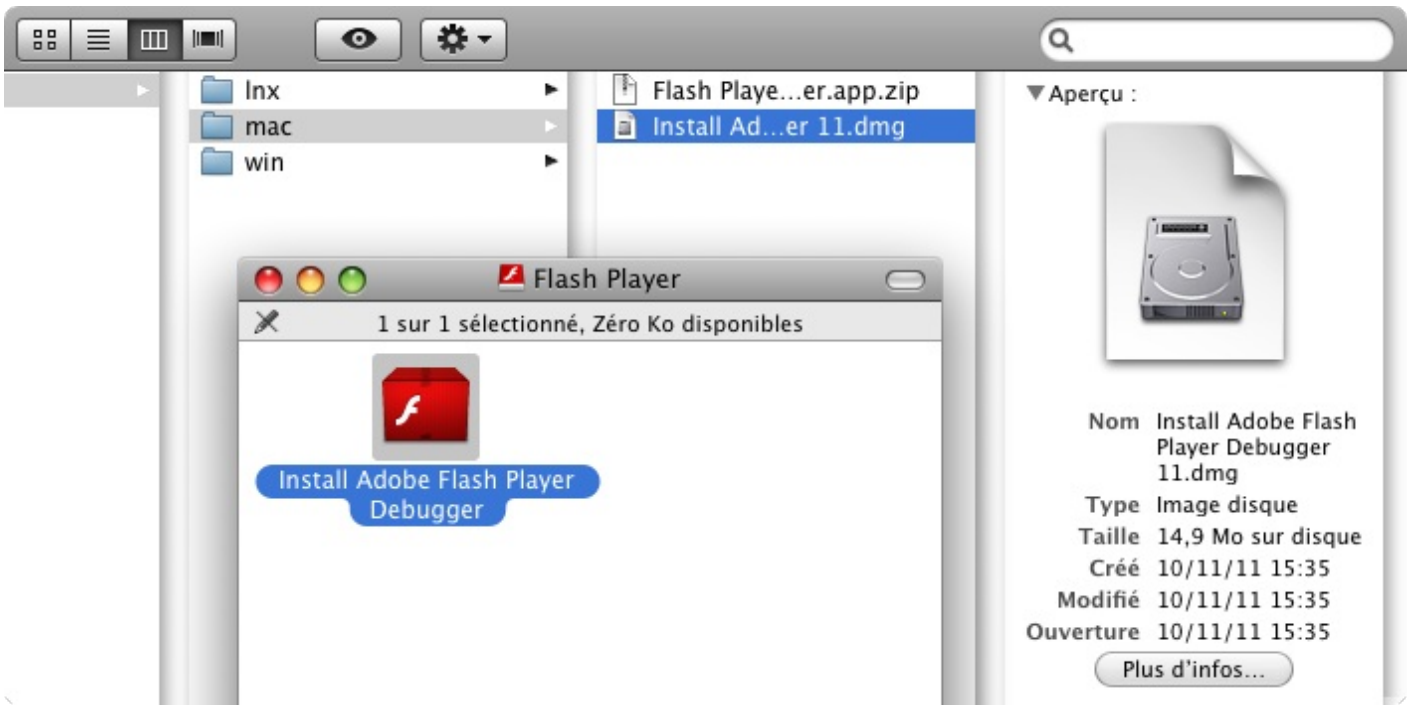
Code : Console

```
sudo mv ~/Flex\ SDK\ 4/runtimes/player/11.1/lnx/flashplayerdebugger /usr/bin/flashp
```

Une fois le déplacement effectué, relancez votre console et replacez-vous dans le dossier `Flex SDK 4/bin`.

Sur Mac

Installez le lecteur Flash en double-cliquant sur l'installateur `Install Adobe Flash Player Debugger 11.dmg` :



Installation de Flash Player Debugger

Créer les sources

Vu que nous n'avons pas encore attaqué l'écriture de programmes en Actionscript 3, je vous ai concocté un petit code de test, qui va nous permettre de vérifier que votre installation fonctionne bien.

Pour les utilisateurs de Windows, il est nécessaire de pouvoir enregistrer vos fichier dans n'importe quelle extension : sinon, il y a de grandes chances que Windows vous trahisse en sauvegardant votre fichier `Test.as` en `Test.as.txt` par exemple, sans que vous ne en rendiez compte ! Pour éviter cela, il faut désactiver une fonctionnalité de Windows qui masque la plupart des extensions : dans une fenêtre de l'explorateur de fichiers, aller dans le menu `Options des dossiers`, puis dans l'onglet `Affichage`, et décochez `Masquer les extensions des fichiers dont le type est connu`.

Commencez par créer un répertoire `Sources` dans le dossier `bin`, où l'on mettra les sources de nos futurs programmes. Dans le répertoire `Sources`, créez un nouveau fichier nommé `Test.as` et copiez-y le code ci-dessous en utilisant un éditeur de texte quelconque, tel que le *Bloc-notes* si vous êtes sous Windows.

Code : Actionscript

```
package
{
    // Programme de test

    // Fichiers nécessaires
    import flash.display.BitmapData;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.filters.BlurFilter;
    import flash.geom.Point;
    import flash.geom.Rectangle;
    import flash.ui.Mouse;
    import flash.display.Bitmap;
    import flash.events.MouseEvent;
    import flash.utils.Timer;

    // Le code
    public class Test extends Sprite {

        private var  bmp:Bitmap;
```

```
        private var _clock:Timer;
        private var _lastPosition:Point;

        public function Test() {
            addEventListener(Event.ADDED_TO_STAGE, Evt_added);
        }

        private function _init():void {
            stage.frameRate = 30;

            _bmp = new Bitmap(new BitmapData(stage.stageWidth,
stage.stageHeight, false, 0x000000));
            _bmp.cacheAsBitmap = true;
            _bmp.opaqueBackground = 0x000000;
            addChild(_bmp);

            _lastPosition = new Point(mouseX, mouseY);

            stage.addEventListener(MouseEvent.MOUSE_MOVE,
Evt_mouseMoved);

            _clock = new Timer(60);
            _clock.addEventListener(TimerEvent.TIMER, Evt_frame);
            _clock.start();
        }

        private function _particle():void {
            var pX:int = mouseX;
            var pY:int = mouseY;

            var x0:int = _lastPosition.x;
            var y0:int = _lastPosition.y;
            var x1:int = pX;
            var y1:int = pY;

            // Tracé de la ligne (Bresenham)

            var error: int;

            var dx: int = x1 - x0;
            var dy: int = y1 - y0;

            var yi: int = 1;
            if( dx < dy ){
                x0 ^= x1; x1 ^= x0; x0 ^= x1;
                y0 ^= y1; y1 ^= y0; y0 ^= y1;
            }
            if( dx < 0 ){
                dx = -dx; yi = -yi;
            }
            if( dy < 0 ){
                dy = -dy; yi = -yi;
            }
            if( dy > dx ){
                error = -( dy >> 1 );
                for ( ; y1 < y0 ; y1++ ) {
                    _bmp.bitmapData.fillRect(new Rectangle(x1 - 4,
y1 - 4, 8, 8), 0xffffff);
                    error += dx;
                    if( error > 0 ){
                        x1 += yi;
                        error -= dy;
                    }
                }
            }else{
                error = -( dx >> 1 );
                for ( ; x0 < x1 ; x0++ ) {
                    _bmp.bitmapData.fillRect(new Rectangle(x0 - 4,
y0 - 4, 8, 8), 0xffffff);
                    error += dy;
```

```
                        if( error > 0 ){
                            y0 += yi;
                            error -= dx;
                        }
                    }
                }

                _lastPosition.x = pX;
                _lastPosition.y = pY;
            }

            private function Evt_added(evt:Event):void {
                removeEventListener(Event.ADDED_TO_STAGE, Evt_added);
                _init();
            }

            private function Evt_mouseMoved(evt:MouseEvent):void {
                _particle();
            }

            private function Evt_frame(evt:TimerEvent):void {
                _bmp.bitmapData.applyFilter(_bmp.bitmapData, new
    Rectangle(0, 0, _bmp.bitmapData.width, _bmp.bitmapData.height), new
    Point(), new BlurFilter(4, 4, 2));
            }
        }
    }
```

Compiler le programme de test

Pour compiler `Test.as`, revenez dans le terminal pour lancer le programme `mxmlc` contenu dans le dossier `bin`.

Windows

Code : Console

```
.\mxmlc.exe "Sources/Test.as"
```

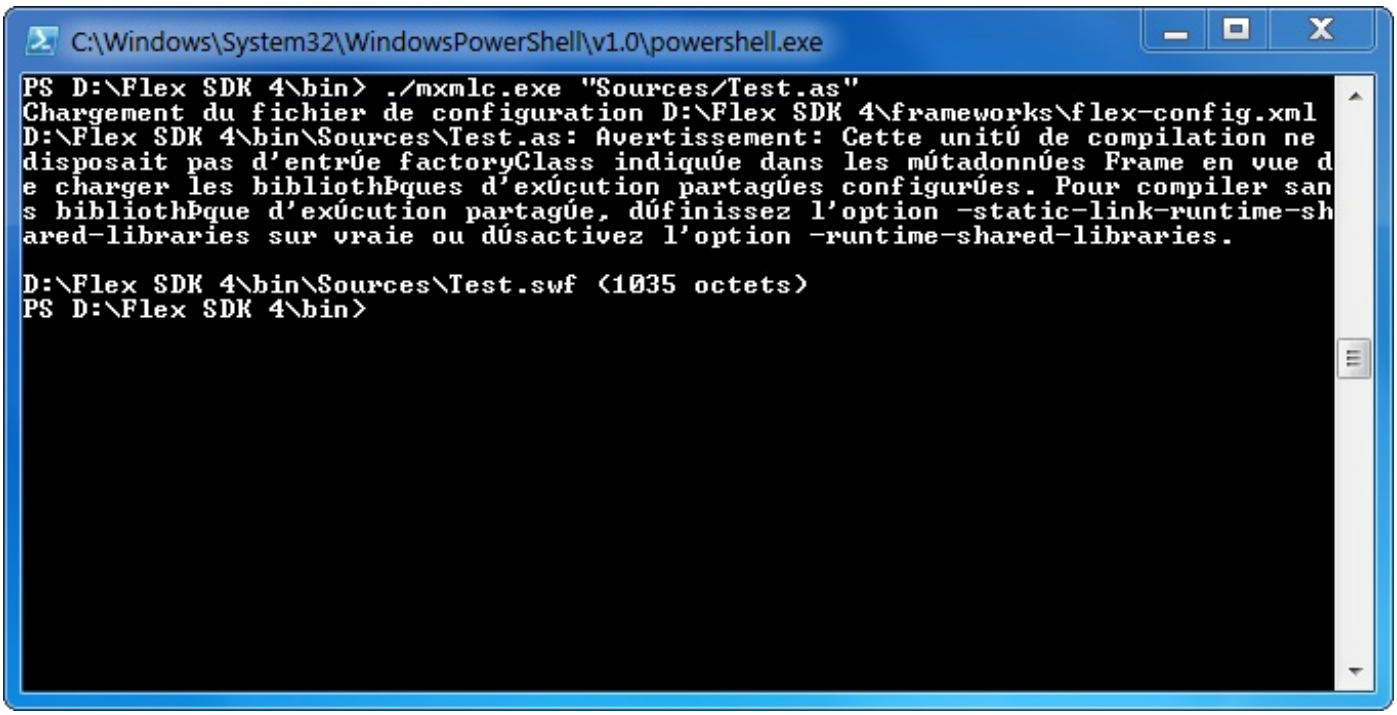
Linux et Max

Code : Console

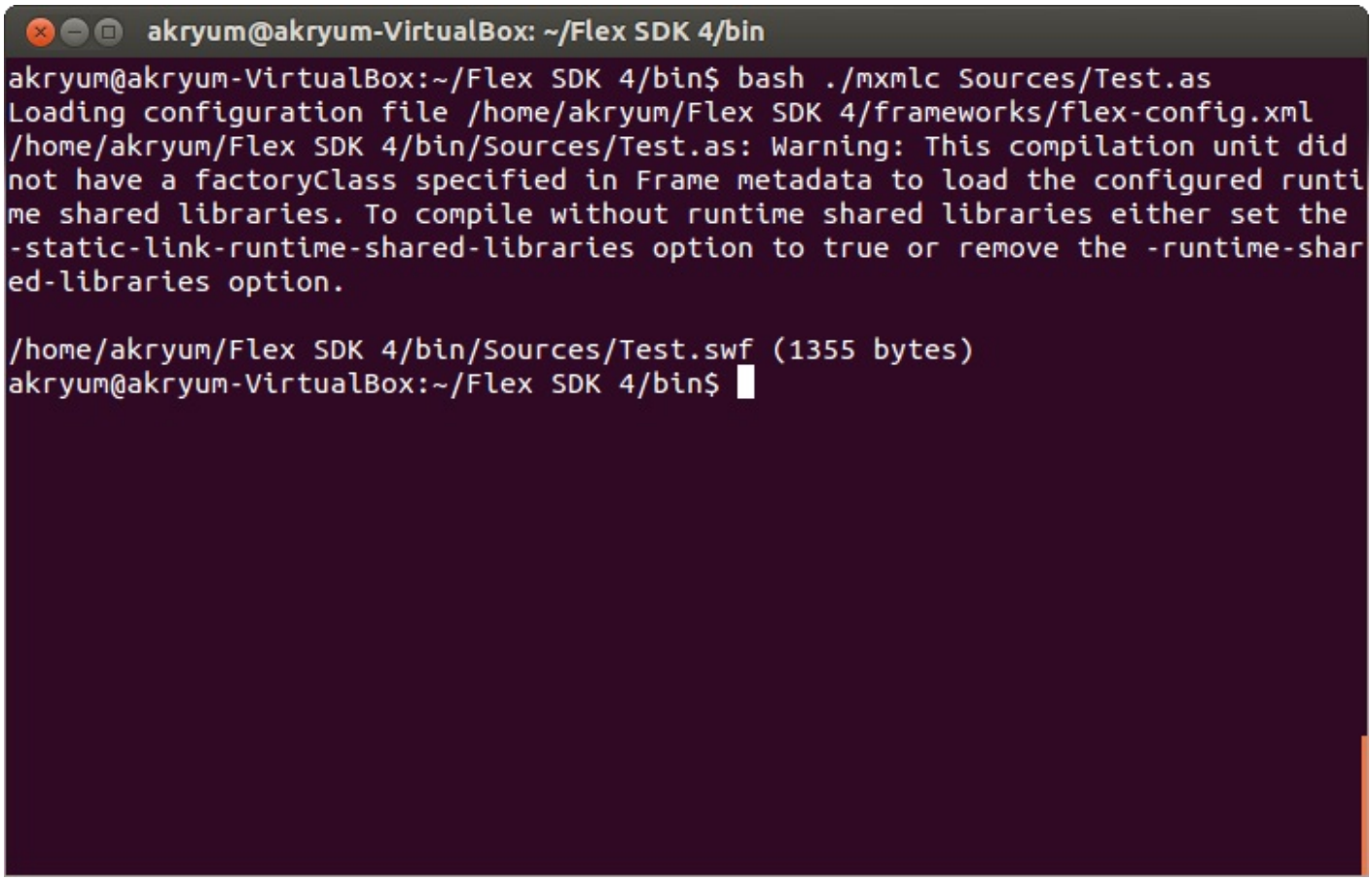
```
./mxmlc Sources/Test.as
```

Il est possible que vous ayez besoin des droits administrateurs sur Linux pour pouvoir compiler. Pour contourner le problème, essayez d'ajouter `bash` ou `sudo` avant la commande `./mxmlc`.

Ensuite, appuyez sur Entrée pour lancer la compilation :



Windows



Linux

Mac

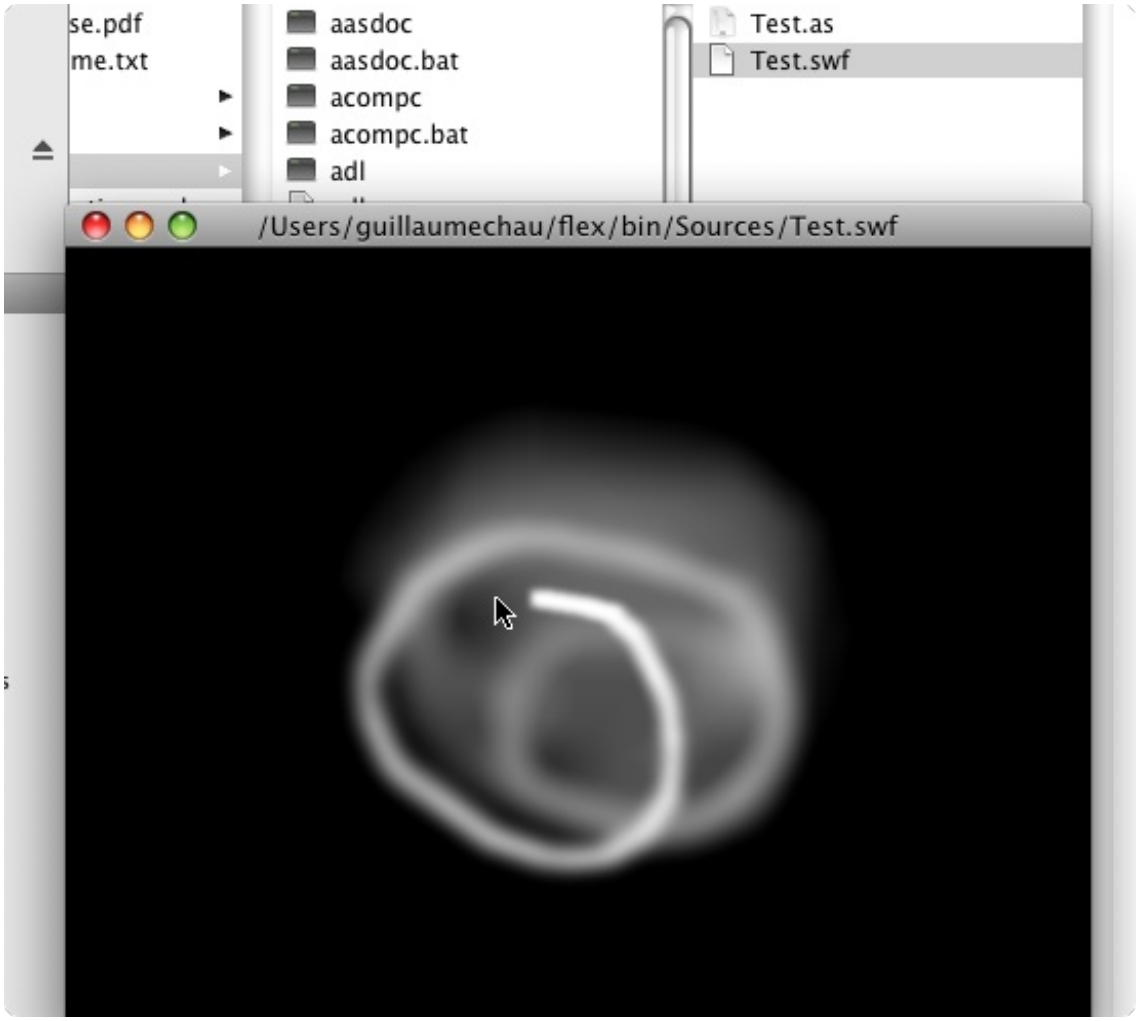
Après quelques instants et si tout s'est bien passé, un fichier `Test.swf` a été créé dans le dossier `Sources` ; il vous suffit de l'ouvrir avec le lecteur de débogage que nous avons téléchargé précédemment !

Sur Windows ou Mac

Double-cliquez sur le fichier `Test.swf`. Si jamais le lecteur Flash ne s'ouvre pas, choisissez-le avec la commande `Ouvrir avec...` ou dans la liste `Sélectionner un programme installé`.


Test.as
22/03/2012 00:43
Fichier AS
3 Ko
Test.swf
22/03/2012 00:43
Fichier SWF
2 Ko
Adobe Flash Player 11
Fichier
Afficher
Contrôle
Aide

Windows



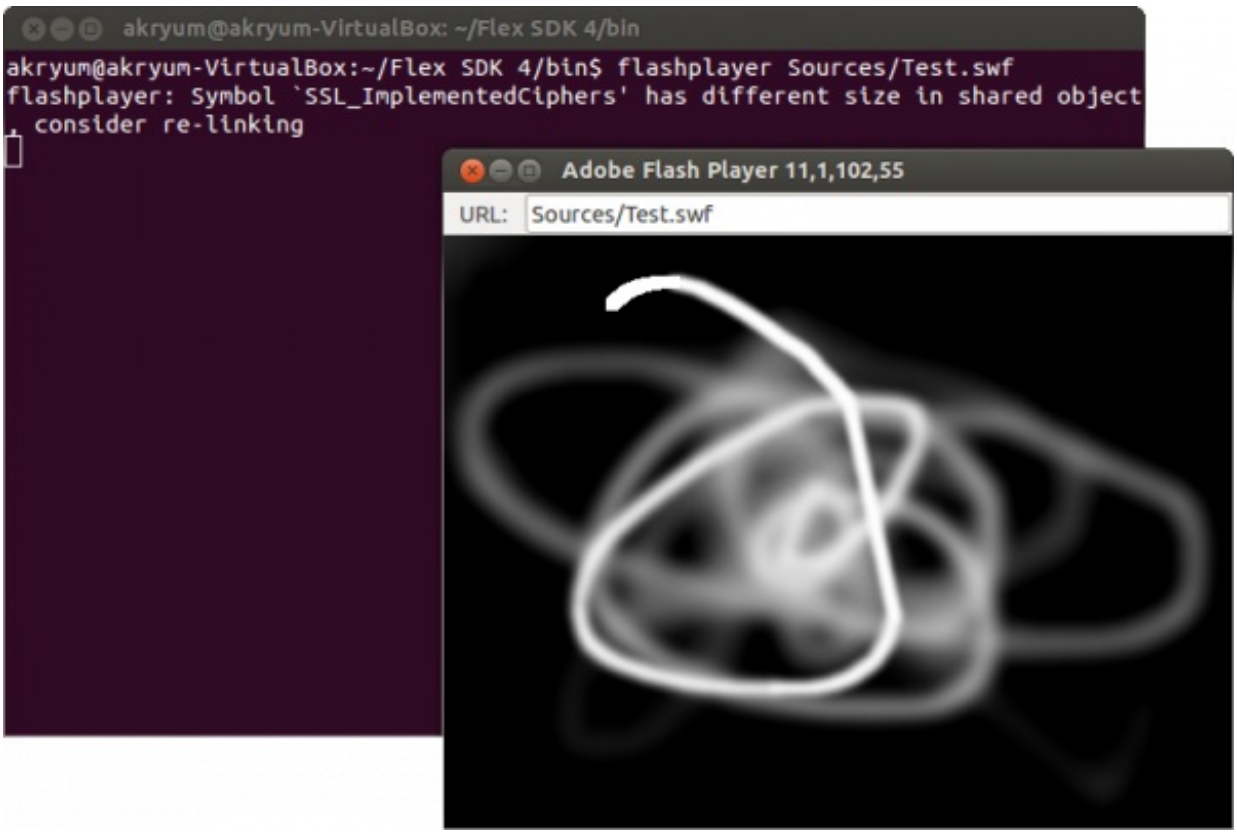
Mac

Sur Linux

Entrez dans votre terminal :

Code : Console

```
flashplayer Sources/Test.swf
```



Linux

Si votre curseur laisse une traînée blanche et brumeuse, cela signifie que la compilation a réussi et que vous êtes prêts pour passer à la suite !

Dis bonjour au monsieur

Nous allons maintenant écrire de A à Z notre premier programme Flash !

Commencez par créer un nouveau fichier Actionscript dans le répertoire `Source`, et nommez-le `Hello.as`.

Structure de notre programme

Un programme écrit intégralement en Actionscript, ce qui sera le cas durant ce cours, présente toujours la même structure : le code est obligatoirement mis dans des **fonctions**, elles-mêmes placées dans une **classe**, englobée par un **paquet** (on utilise alors le mot anglais **package**).
Pour l'instant, il y a de grandes chances que les trois termes que je viens d'utiliser soient obscurs pour vous ; la seule chose importante à retenir est l'organisation de ces trois niveaux qui composent le code source de tout fichier Actionscript. Pas de panique : nous explorerons plus en détails chacune de ces notions plus tard dans le cours.

Voici un petit schéma récapitulatif :

Structure d'un programme

Le package

Commençons par le plus simple : le `package`. Il est composé d'un nom facultatif et d'un bloc d'accolades. Nous utilisons le **mot-clé** `package` comme ceci :

Code : Actionscript

```
package nom {

}
```

Un **mot-clé** (ou réservé) est une expression qui est utilisée par le langage et qui lui est donc réservée. Vous ne pourrez donc pas utiliser ce type de mots autrement que tels qu'ils ont été définis en Actionscript.

Le nom du package décrit la position de la classe dans l'arborescence des fichiers de votre projet. Effectivement, vous ne pouvez choisir vous-mêmes le nom du package : vous devez mettre le chemin relatif du fichier par rapport au fichier principal (celui que l'on compile avec la commande `mxmlc` vue plus haut), en séparant les dossiers par des points (à la place des slashes ou antislashes).

Par exemple, si votre fichier principal `Hello.as` se trouve dans le dossier `source`, et que vous créez un nouveau dossier `ET` dans lequel vous rajoutez un dossier `telephoner`, en y mettant un fichier Actionscript `Maison.as` qui est utilisé par votre programme principal, vous devrez écrire son package ainsi :

Code : Actionscript

```
package ET.telephoner {

}
```

L'arborescence des fichiers est alors `Sources/ET/telephoner/Maison.as`.

Comment cela, on peut utiliser un autre fichier Actionscript depuis notre programme principal ?

Bien sûr ! C'est même très utile : il vaut mieux faire plusieurs fichiers bien triés (un dossier pour les utilitaires, un dossier pour le chargement de son...), que tout mettre dans un seul et unique fichier. Imaginez que votre programme fasse plusieurs centaines de milliers de lignes de code : il vous sera impossible de vous y retrouver !

Nous allons voir cette notion appelée *importation de fichiers* dans très peu de temps.

Qu'en est-il de notre fichier Hello.as ?

Très bonne question : pour le programme principal contenu dans le fichier que vous allez compiler avec la commande `mxmlc`, il n'y a pas de nom de package ! Et oui, il est déjà dans le dossier principal de votre projet, inutile donc de mettre un chemin.

Dans ce cas, le package s'écrit ainsi :

Code : Actionscript

```
package {

}
```

Vous pouvez écrire ce code dans votre fichier vide si vous ne l'avez pas déjà fait !

Importer des fichiers Actionscript

Comme nous l'avons vu, importer des fichiers peut s'avérer très utile, voire vital dans la plupart des cas. L'importation est une **instruction** (ou une ligne de code si vous préférez) qui permet de rendre tout le code du fichier importé utilisable dans notre code. Il y a deux types de fichiers que vous pouvez importer : les fichiers contenus dans les packages de Flash (commençant par `flash.`), et vos propres fichiers. Dans les deux cas, on utilise le mot-clé **`import`** suivi du package du fichier à importer, son nom et un point-virgule :

Code : Actionscript

```
import ET.telephone.Maison;
```

La plupart des instructions et lignes de code devraient être terminées par un point-virgule en fin de ligne. Je dis « devraient », car il est toléré d'omettre le point-virgule, mais c'est très fortement déconseillé : il est le symbole qui indique clairement la fin de votre ligne de code, et il est possible dans de rares cas que vous obteniez des erreurs à cause de cela.
Votre ordinateur ne s'autodétruira pas si vous ne mettez pas les points-virgules, mais ce serait une très, très, très mauvaise habitude : si jamais vous voulez apprendre et utiliser un autre langage, vous oublierez constamment les points-virgules et maudirez le jour où vous avez décidé de ne pas suivre mon conseil...
Vous remarquerez qu'il ne faut pas mettre de point-virgule quand il s'agit de blocs de code délimités par des accolades ({ et }), comme pour le package que l'on vient de voir.

Les importations s'écrivent juste après avoir ouvert la première accolade du package :

Code : Actionscript

```
package {

    import ET.telephone.Maison;

}
```

Pour notre programme, il va falloir importer un fichier *standard* de Flash : `Sprite`. Il est nécessaire pour toute application : nous détaillerons son utilité bien plus tard dans le cours, car il nous manque trop de notions actuellement.
Je peux tout de même vous dire que cela nous permet d'afficher le programme, et donc de le lancer : effectivement, tout programme Flash a besoin d'un affichage pour s'exécuter.

Je vous donne le code du fichier importé que vous devez avoir dans votre `Hello.as` :

Code : Actionscript

```
package {

    import flash.display.Sprite;

}
```

Vous l'aurez sûrement compris, le package du fichier est `flash.display` (gestion de l'affichage).

La classe

Ensuite, vient la... Comment ça vous ne savez pas ? Retournez vite regarder le schéma juste au-dessus !
Effectivement, dans le package, on trouve une *classe*. Cette notion est un concept très important de la programmation *orientée objet* que j'ai mentionné dans l'introduction. Encore une fois, il est trop tôt pour développer cette notion, mais rassurez-vous : une partie entière de ce cours est consacrée à l'*orienté objet*.

L'essentiel est de retenir que la classe est obligatoire et que sa syntaxe est la suivante : les mots-clés **`public class`** suivis du nom de la classe et d'un bloc d'accolades, comme pour le package.

Code : Actionscript

```
public class NomDeMaClasse {

}
```

Attention ! Le nom d'une classe ne doit contenir *que des lettres*, et doit être *identique au nom du fichier* dans lequel elle se trouve (sans l'extension). De plus, il doit impérativement *commencer par une majuscule* : cela explique pourquoi tous nos fichiers Actionscript ont une majuscule depuis le début du cours !

Étant donné que le nom du fichier est le même que celui de la classe de ce fichier, nous parlerons désormais de *classe* dans les deux cas.

Il existe une notation appelée *Camel* (chameau) ou *CamelCase* (notation chameau), très utilisée dans le monde de l'informatique. Je l'ai utilisée pour écrire `NomDeMaClasse` : chaque mot débute par une majuscule, comme des bosses de chameau ! Son utilité est purement esthétique, car cette notation améliore la lisibilité du code lorsque l'on ne peut pas utiliser d'espaces ou d'autres caractères spéciaux. C'est le cas ici avec le nom des classes.
Je pense que c'est une bonne habitude à prendre, alors autant commencer tout de suite : avouez que si j'avais écrit `Nomdemaclasse`, ce serait beaucoup moins lisible !

La classe se dispose juste après les importations de fichiers, comme ceci, pour notre fichier `Hello.as` :

Code : Actionscript

```
package {

    import flash.display.Sprite;

    public class Hello extends Sprite {

    }
}
```

Les plus observateurs d'entre vous auront remarqué que j'ai ajouté deux mots derrière le nom de notre classe. Le mot-clé **`extends`** (étendre en anglais) permet d'utiliser la classe `Sprite` d'une manière un peu spéciale, comme nous le verrons dans la partie consacrée à l'orienté objet.

Les fonctions

Maintenant que notre classe est prête, il faut écrire une ou plusieurs fonctions pour pouvoir mettre du code dedans !
Une de ces fonctions est obligatoire, et elle porte un nom : il s'agit du **constructeur** de la classe. Devinez quoi ? C'est encore lié à l'orienté objet !
Pour faire simple, ce constructeur est automatiquement exécuté dans certains cas, lorsque l'on utilise une classe. Ici, le code à l'intérieur sera parcouru au lancement de notre programme. Pour écrire un constructeur, on utilisera les mots-clés **`public function`** suivis du nom de la classe, de parenthèses et d'un nouveau bloc d'accolades.

Pour notre classe `Hello` contenue dans notre fichier `Hello.as`, le constructeur ressemblera à ceci :

Code : Actionscript

```
public function Hello(){

}
```

Et voilà ! Notre fichier est fin prêt pour que nous commencions (enfin) à coder !

Voici le code complet de notre fichier `Hello.as` pour que vous puissiez vérifier :

Code : Actionscript

```
// Premier niveau : le paquet
package {
```

```
        // Les classes importées
        import flash.display.Sprite;

        // Deuxième niveau : la classe du fichier
        public class Hello extends Sprite {

            // Troisième niveau : la ou les fonctions

            // Constructeur de la classe
            public function Hello() {

                // Nous allons coder ici !

            }

        }
    }
```

Commentez votre code !

Avez-vous remarqué que j'ai inséré du texte en français dans le code précédent, qui n'a a priori rien à faire là ?
On appelle cela des commentaires. Vous pouvez y écrire absolument n'importe quoi, et pour cause : ils seront tout simplement ignorés lorsque vous lancerez votre programme.
Vous vous demandez alors à quoi servent-ils ? La réponse est plutôt simple : commenter un code permet de s'y retrouver, même si nous avons nous-mêmes écrit le code en question. Imaginez que vous reveniez dessus quelques mois plus tard : s'il n'y a aucun commentaire, vous serez aussi perdu que si ce n'était pas vous le programmeur ! Cela peut même arriver dans un intervalle de quelques jours seulement. Et ce serait encore pire si vous souhaitiez que votre code soit lu ou utilisé par d'autres personnes...
Mais attention, il ne s'agit pas non plus d'inonder votre code dans les commentaires : vous ne réussiriez qu'à le rendre encore plus illisible et incompréhensible ! Il faut trouver un juste milieu : commentez quand cela est nécessaire, pour décrire brièvement ce que fait un bout de votre programme, afin de vous en rappeler facilement plus tard.

Il existe deux types de commentaires : les commentaires en ligne et les commentaires multi-lignes.

Les commentaires en ligne

Ce sont des commentaires qui ne comportent qu'une seule ligne. On les débute par deux slashes, comme ceci :

Code : Actionscript

```
// Voici un commentaire en ligne
```

Tout le texte suivant les deux slashes sera considéré comme du commentaire.

Vous pouvez mettre un commentaire en fin de ligne, sans gêner personne :

Code : Actionscript

```
import display.Sprite; // Pour l'affichage
```

Les commentaires multi-lignes

Cette fois-ci, il est possible d'étaler notre commentaire sur plusieurs lignes. Pour cela, il faut débuter notre commentaire par un slash et une astérisque, et terminer explicitement le commentaire par une astérisque et un slash :

Code : Actionscript

```
/* Ceci
```

```
est un commentaire
sur quatre
lignes. */
```

Afficher un message dans la console

Avant toute chose, il est important de souligner que la technologie Flash n'a jamais été conçue pour être utilisée en lignes de commande : en effet, son objectif est de proposer des applications et des animations entièrement graphiques. Cela implique que l'on ne puisse qu'afficher du texte à la console (avec quelques efforts), et qu'il est impossible d'entrer des données au clavier via la console, comme en langage C par exemple.

C'est une fonction qui nous permettra d'afficher des messages dans la console : elle répond au doux nom de `trace()`, et est quasiment exclusivement utilisée pour le débogage des programmes Flash.

Cette fonction est accessible partout dans tout code Actioncript, sans rien à faire d'autre que de l'*appeler*.

L'appeler ? Comme mon chien pour partir en balade ?

Oui, on utilise le terme *appeler* lorsque l'on utilise une fonction : nous n'allons pas écrire du code entre les accolades de la fonction `trace()`, nous allons nous contenter de l'utiliser. En effet, son code est déjà prêt et fait partie du Lecteur Flash lui-même !

Pour l'utiliser, nous allons prendre presque la même syntaxe que celle du constructeur `Hello` (qui est lui aussi une fonction), sans le bloc d'accolades, et sans oublier le point-virgule (car il s'agit d'une instruction) :

Code : Actionscript

```
trace("texte à afficher");
```

Entre les deux parenthèses, vous devez spécifier à la fonction `trace()` ce qu'elle doit afficher. Ce peut être du texte (délimité par des guillemets) ou un nombre.
Par exemple, pour afficher 42 dans la console, nous écririons :

Code : Actionscript

```
trace(42);
```

À vous de jouer ! Faites en sorte sans le tester que votre programme `Hello` soit capable d'afficher le grand classique « Hello world ! ». Souvenez-vous où il faut mettre les instructions dans le programme `Hello` que nous avons écrit jusqu'à présent !

Code : Actionscript

```
// Premier niveau : le paquet
package {

    // Les classes importées
    import flash.display.Sprite;

    // Deuxième niveau : la classe du fichier
    public class Hello extends Sprite {

        // Troisième niveau : la ou les fonctions

        // Constructeur de la classe
        public function Hello() {
```

```
                trace("Hello world !");

            }

        }
    }
```

Place au test !

Compiler notre programme

Commençons tout d'abord par compiler `Hello.as`. Il y a une petite différence par rapport à tout à l'heure : cette fois-ci, il va falloir activer le mode débogage lors de la compilation, pour que l'on puisse afficher le message « Hello world ! » dans la console. Pour ce faire, ajoutons un paramètre lors de la compilation : `-debug=true`.

Sur Windows :

Code : Console

```
.\mxmlc.exe -debug=true "Sources/Hello.as"
```

Sur Linux :

Code : Console

```
./mxmlc -debug=true "Sources/Hello.as"
```

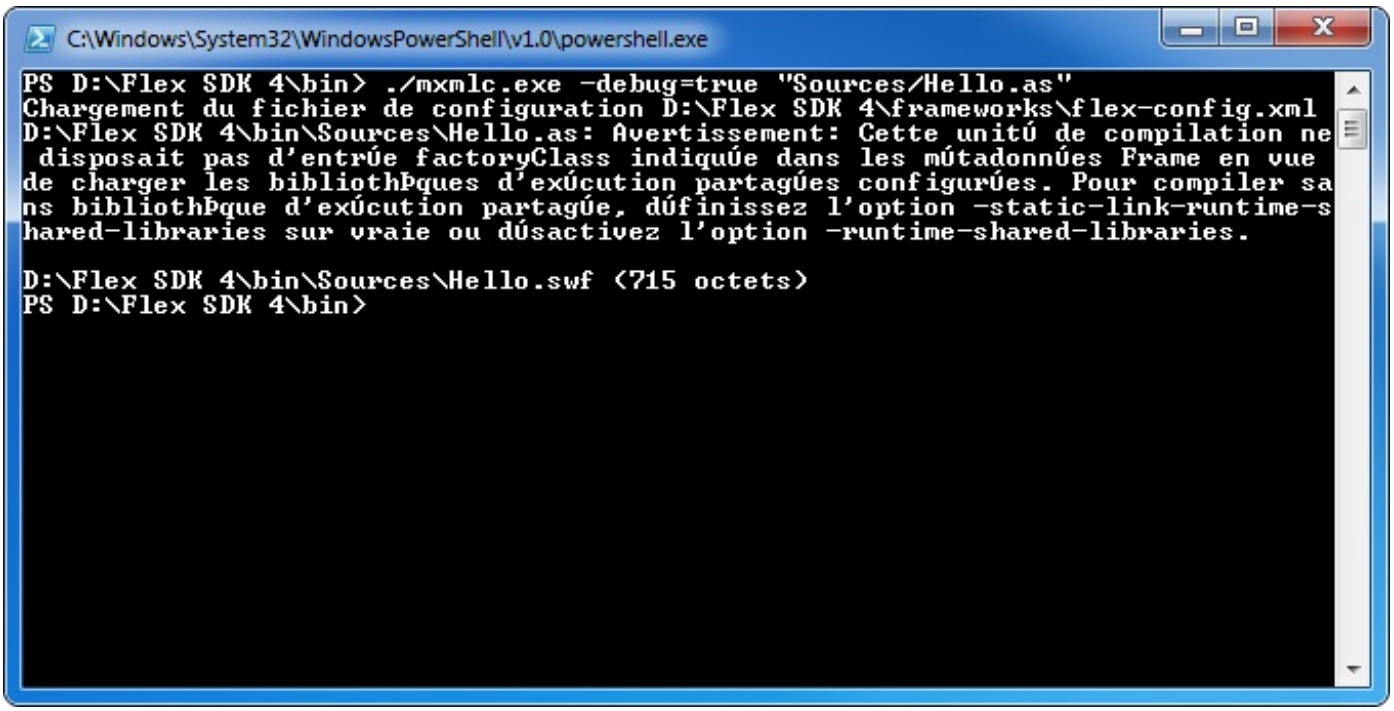
Et enfin, sur Mac :

Code : Console

```
./mxmlc -debug=true Sources/Hello.as
```

Encore une fois, il est possible que vous ayez besoin des droits administrateurs sur Linux pour pouvoir compiler. Pour contourner le problème, essayez d'ajouter `bash` ou `sudo` avant la commande `./mxmlc`.

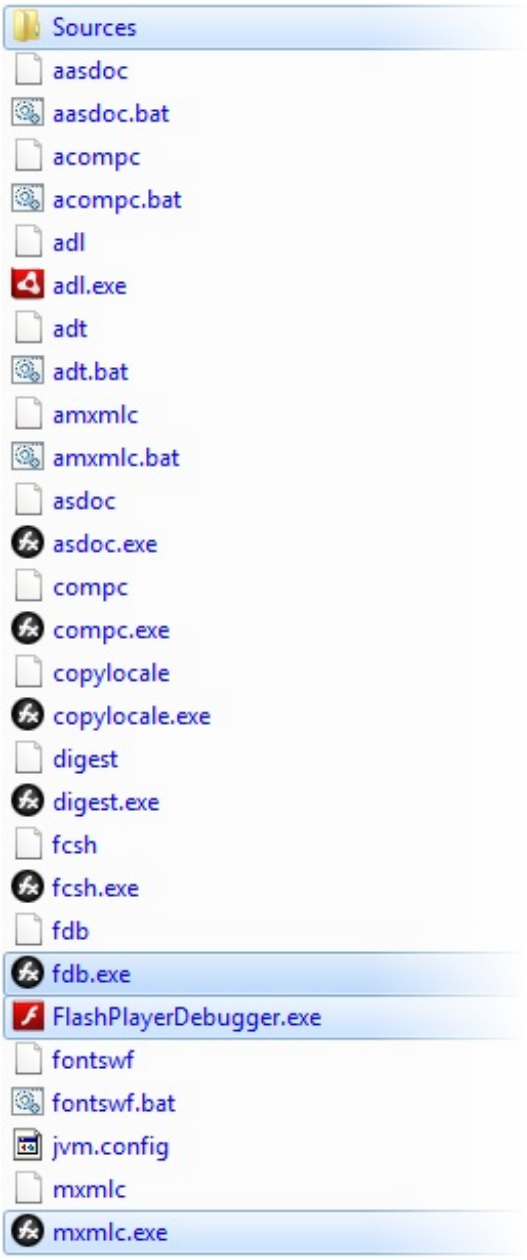
Si tout c'est bien passé, votre console devrait afficher quelque chose comme ceci :



Résultat de la compilation

Lancer le débogueur Flash

Pour pouvoir afficher les messages de la fonction `trace()`, il nous faut utiliser un autre outil à notre disposition : le débogueur Flash. En effet, Flash Player, même en version de débogage, ne peut pas afficher des messages en continu dans une console. Il se contente de les envoyer : le débogueur va alors se charger de récupérer les messages et de nous les présenter dans la console.
Il s'agit du programme `fdb` qui devrait être dans le répertoire `Flex SDK 4/bin` :



Contenu du répertoire Flex SDK 4/bin

Lancez le programme dans votre console :

Sur Windows :

Code : Console

```
.\fdb.exe
```

Sur linux :

Code : Console

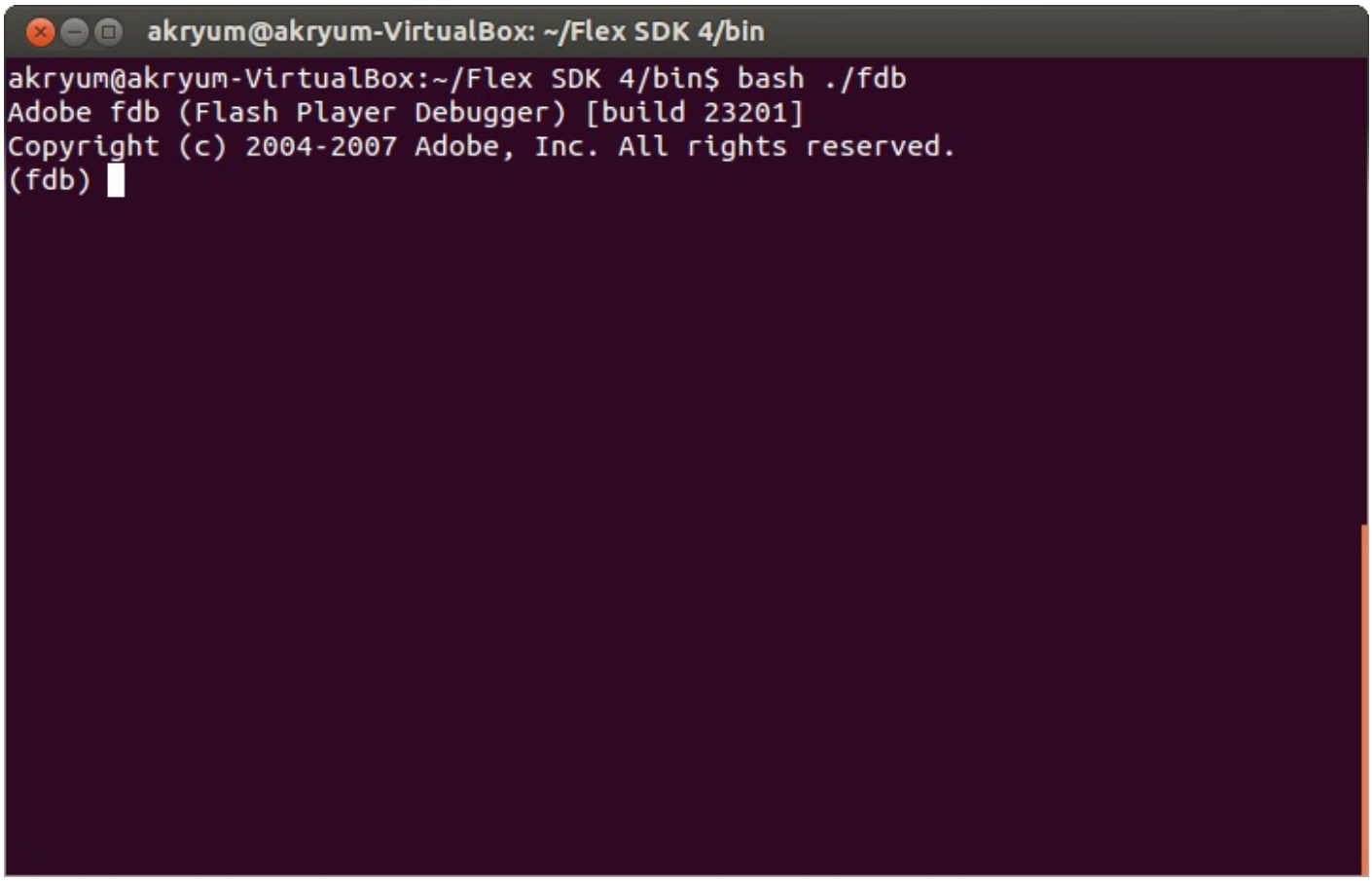
```
./fdb
```

Sur Mac :

Code : Console

```
./fdb
```

Un nouvel invité de commande, géré par `fdb` devrait apparaître :



Le nouvel invité de commande géré par fdb

Lancer notre programme

Tout d'abord il faut lancer une session de débogage, pendant laquelle le débogueur essayera de récupérer les messages de débogage du Lecteur Flash :

Code : Console

```
run
```

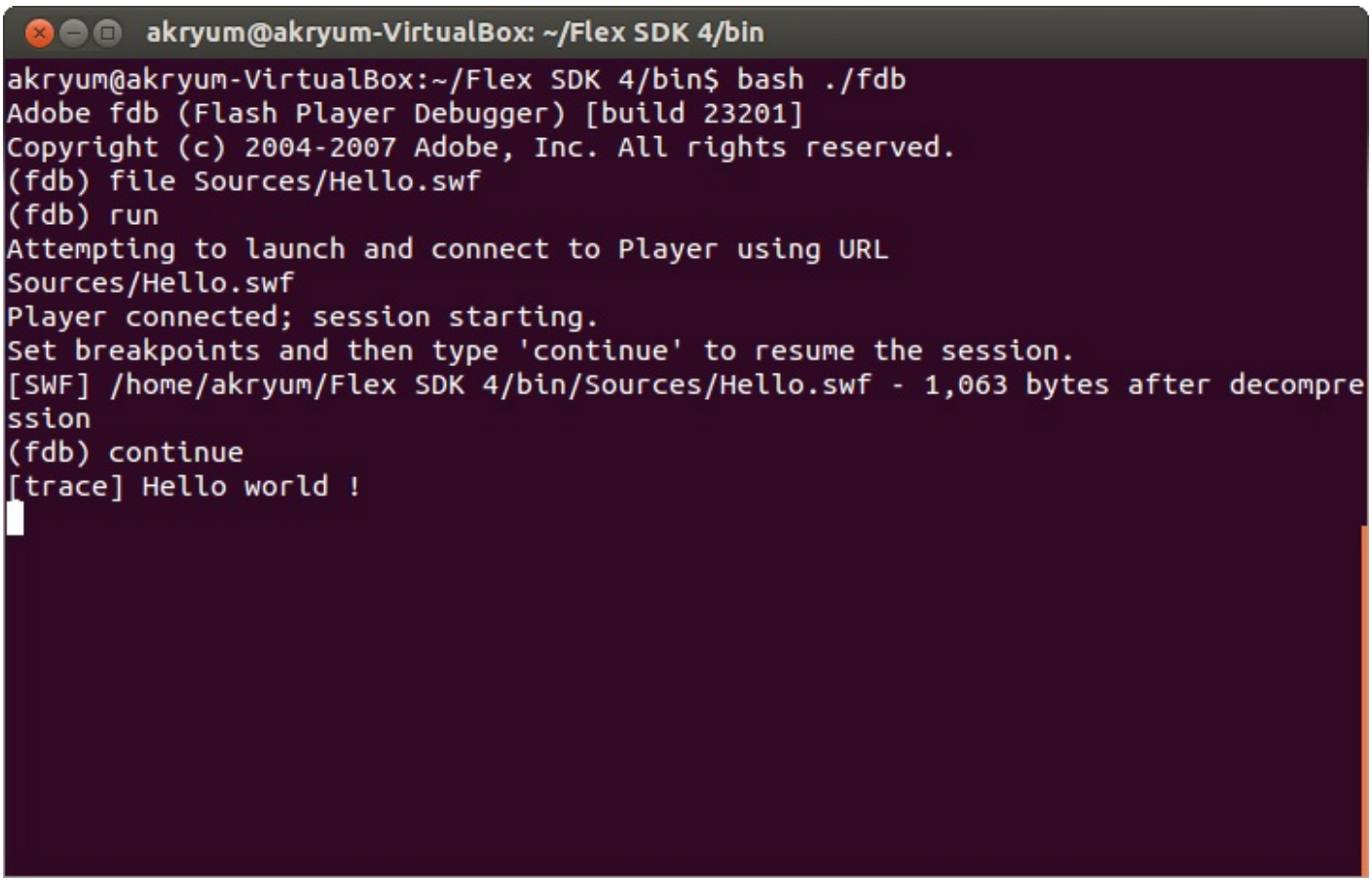
Le débogueur est ainsi en attente du Lecteur Flash : lancez votre programme en double-cliquant dessus, ou avec la commande que l'on a vu plus haut dans le cas de Linux. Le Lecteur Flash n'est cependant pas encore démarré, il ne devrait donc pas se passer grand-chose à l'écran.

Une fois que la session est correctement démarrée, entrez cette commande dans `fdb` pour réellement lancer le programme :

Code : Console

```
continue
```

Miracle ! Notre `trace("Hello world !");` a fonctionné, et le message est apparu dans la console !



Résultat de la fonction trace()

Une fenêtre blanche s'est ouverte : il s'agit de notre animation ! Pour l'instant, elle ne fait presque rien : nous remédierons à cela dans la suite du cours.

Vous pouvez maintenant fermer votre programme. Si plus tard vous voulez quitter `fdb`, entrez cette commande :

Code : Console

```
quit
```

Vous pouvez également lister l'intégralité des commandes du débogueur avec cette commande :

Code : Console

```
help
```

Lorsque vous programmez en lignes de commande, je vous conseille d'ouvrir deux consoles : une pour la compilation à l'aide de `mxmlc`, et une autre pour le débogueur `fdb`. Cela vous évitera de quitter et relancer ce dernier à chaque compilation ; il vous suffira d'utiliser la commande `run` à chaque fois que vous voulez tester votre programme.

Il en faut des efforts pour déboguer en lignes de commande, mais une fois que vous avez pris le coup de main, c'est plus facile.
Mais pour se faciliter la vie, mieux vaut s'équiper avec les meilleurs outils ! En effet, il existe des éditeurs de code très facile d'utilisation qui intègrent également des outils prêts à l'emploi pour compiler et tester nos programmes !

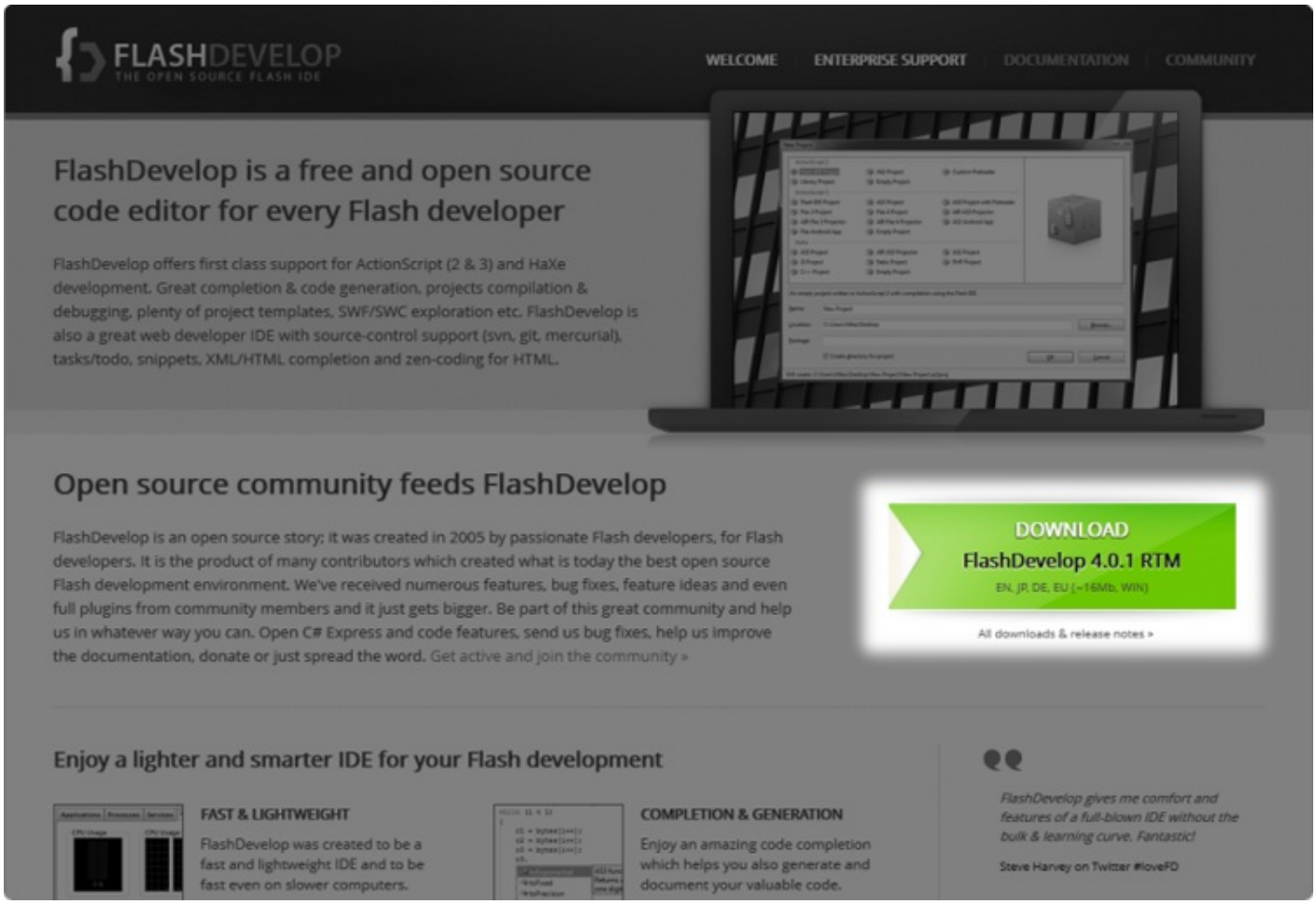
FlashDevelop à la rescousse !

Pour Windows, il existe un très puissant éditeur pour programmer en Flash, et de plus, il est gratuit et libre : j'ai nommé

Flashdevelop !
Flashdevelop est ce que l'on appelle un *IDE* (*Integrated Development Environment*, environnement de développement intégré en anglais) : c'est un logiciel contenant tout le nécessaire pour programmer dans un ou plusieurs langages (ici l'Actionscript, mais pas seulement). Vous allez voir : compiler sera désormais aussi simple que d'appuyer sur un bouton !

Téléchargement

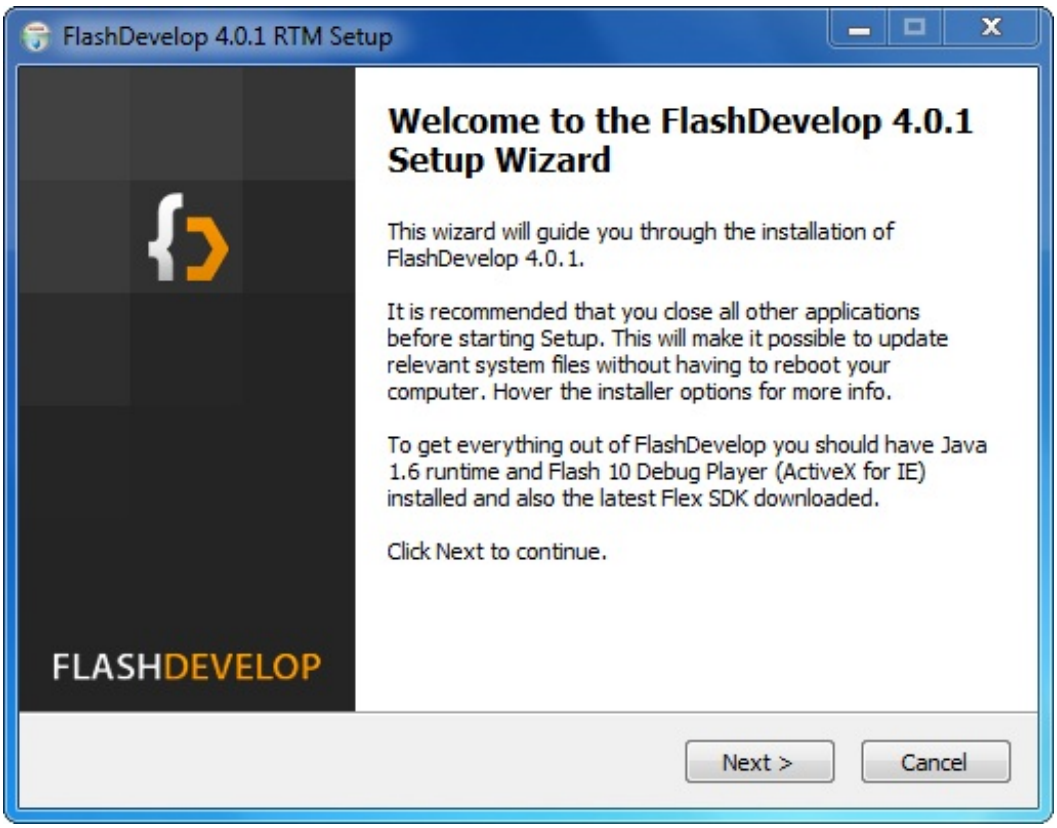
Flashdevelop est donc disponible au téléchargement gratuitement, mais malheureusement la version officielle est réservée aux utilisateurs de Windows. Commencez par télécharger Flashdevelop à cette adresse. Il est possible, pour les utilisateurs d'un autre système d'exploitation d'utiliser une machine virtuelle, tant que les développeurs n'auront pas eu le temps d'adapter Flashdevelop à d'autres supports. Il existe également une alternative, sous forme d'un plugin pour le logiciel Eclipse. Rassurez-vous, le reste du cours ne porte pas sur Flashdevelop, vous pourrez tout faire sans ! 



Téléchargement de FlashDevelop

Installation

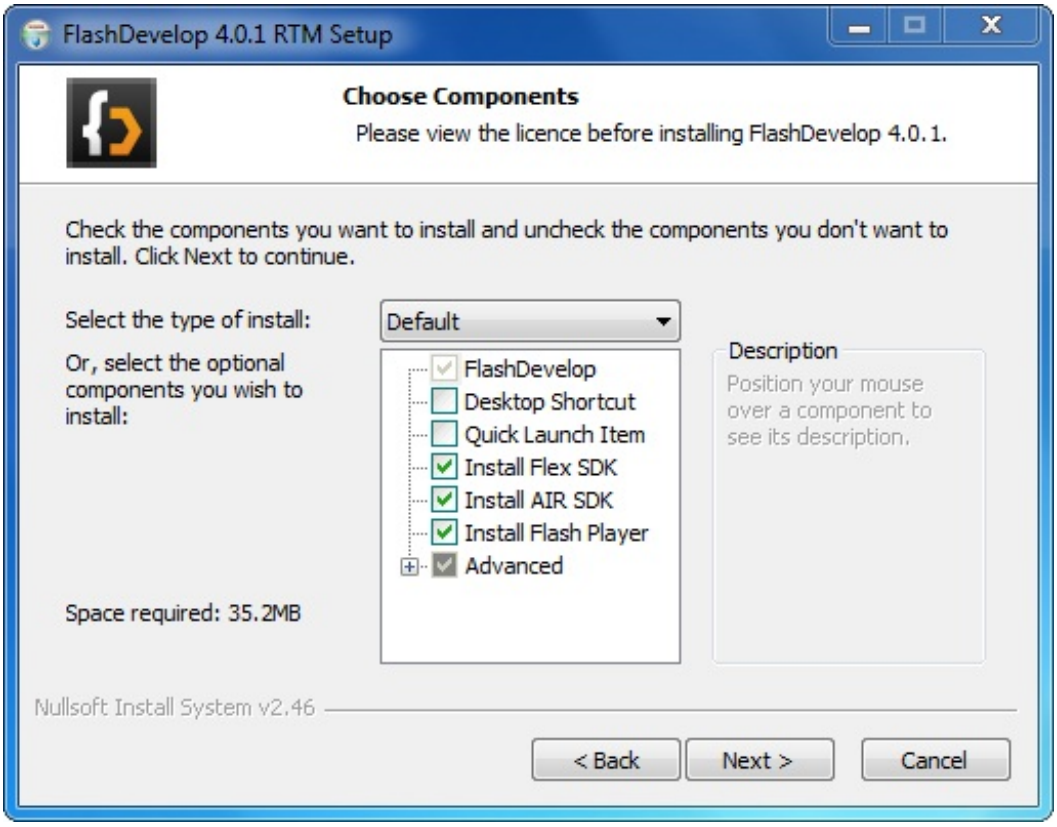
Une fois l'installateur de Flashdevelop téléchargé, lancez-le et appuyez sur `Next`.



Installation de FlashDevelop :

étape 1

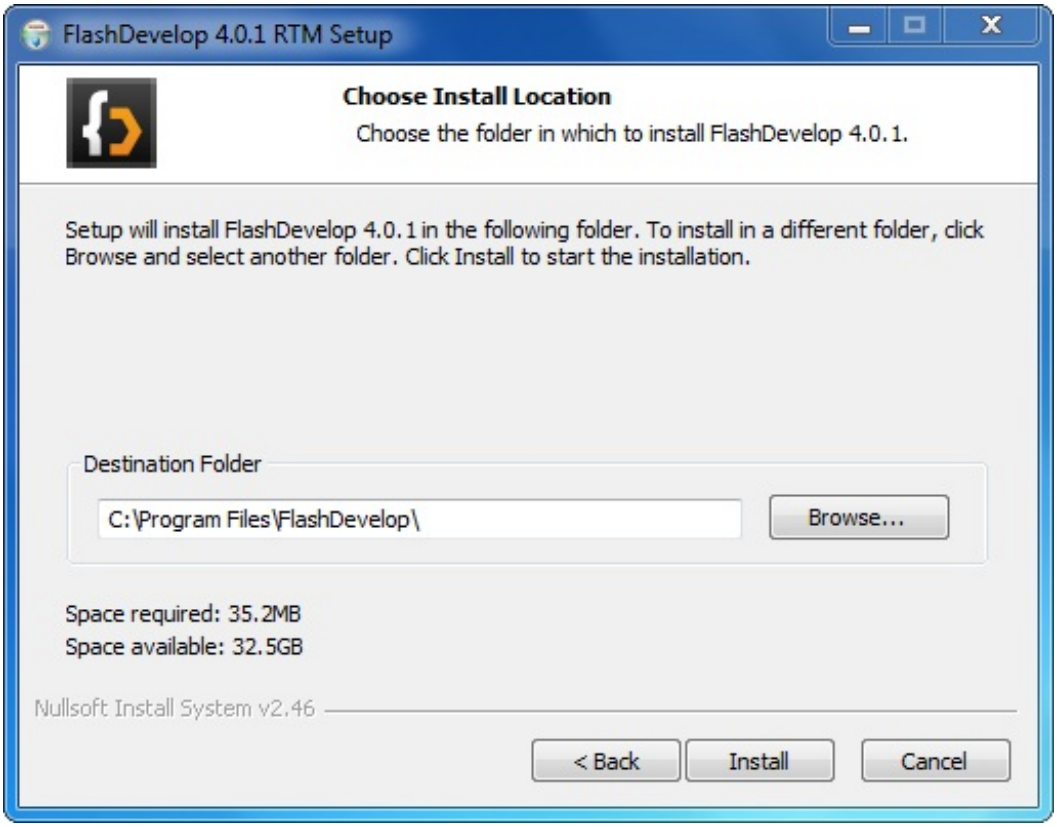
Cet écran nous permet de choisir d'installer ou non des composants supplémentaires, comme le *Flex SDK*. Pour éviter tout problème, nous allons laisser les options par défaut, mais vous pouvez décocher `Install Flex SDK` si vous l'avez déjà téléchargé et que vous êtes sûr de vous. Je vous conseille toutefois de laisser le programme installer le compilateur lui-même, pour éviter tout problème. Cliquez sur `Next`.



Installation de FlashDevelop :

étape 2

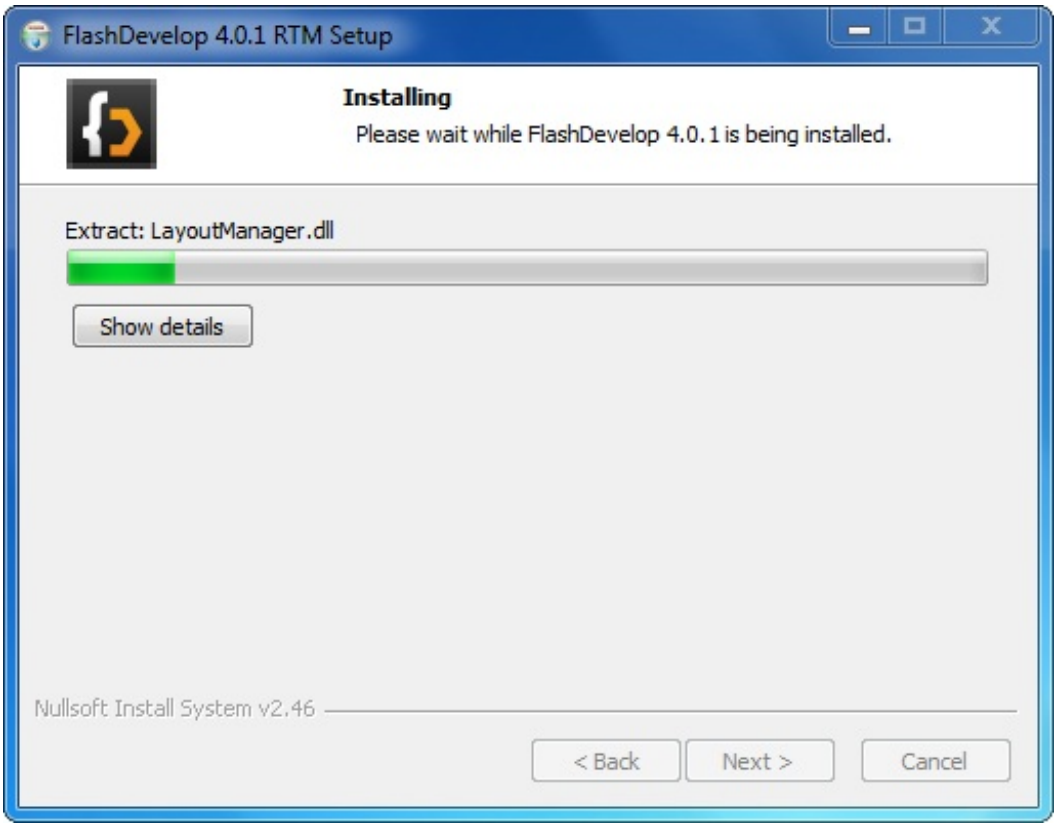
Choisissez le dossier d'installation de Flashdevelop, puis cliquez une nouvelle fois sur `Next`.



Installation de FlashDevelop :

étape 3

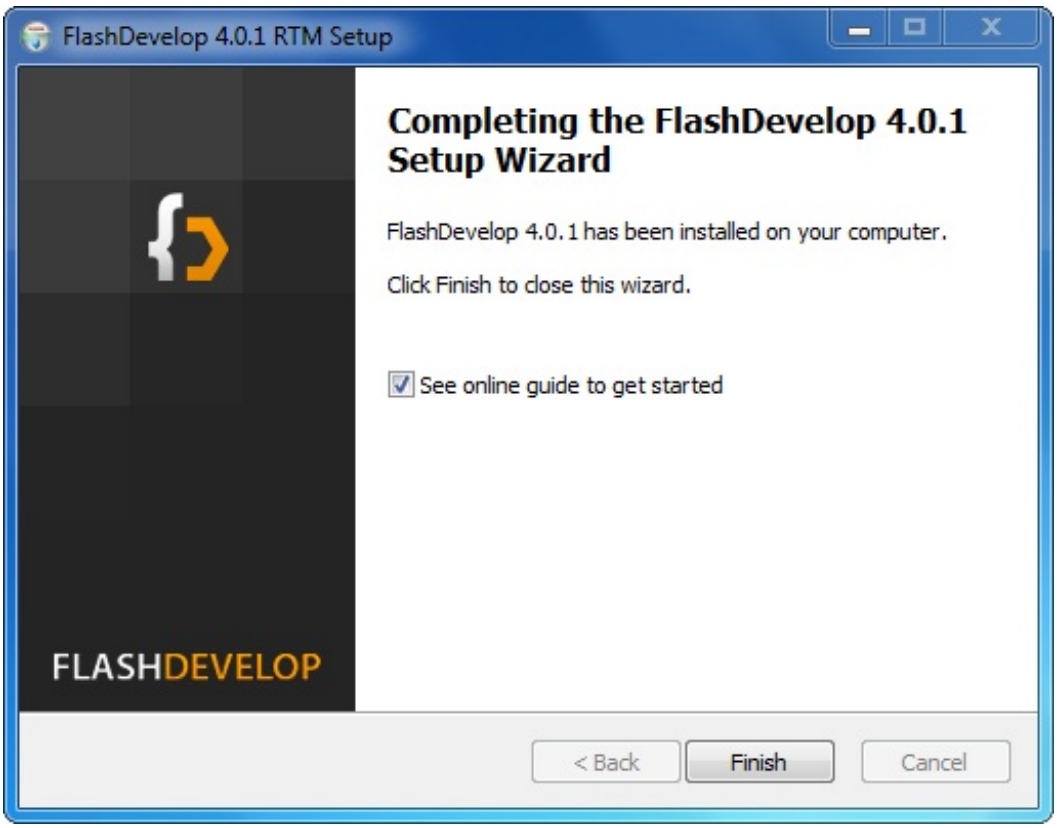
Allez prendre un café pendant l'installation !



Installation de FlashDevelop :

étape 4

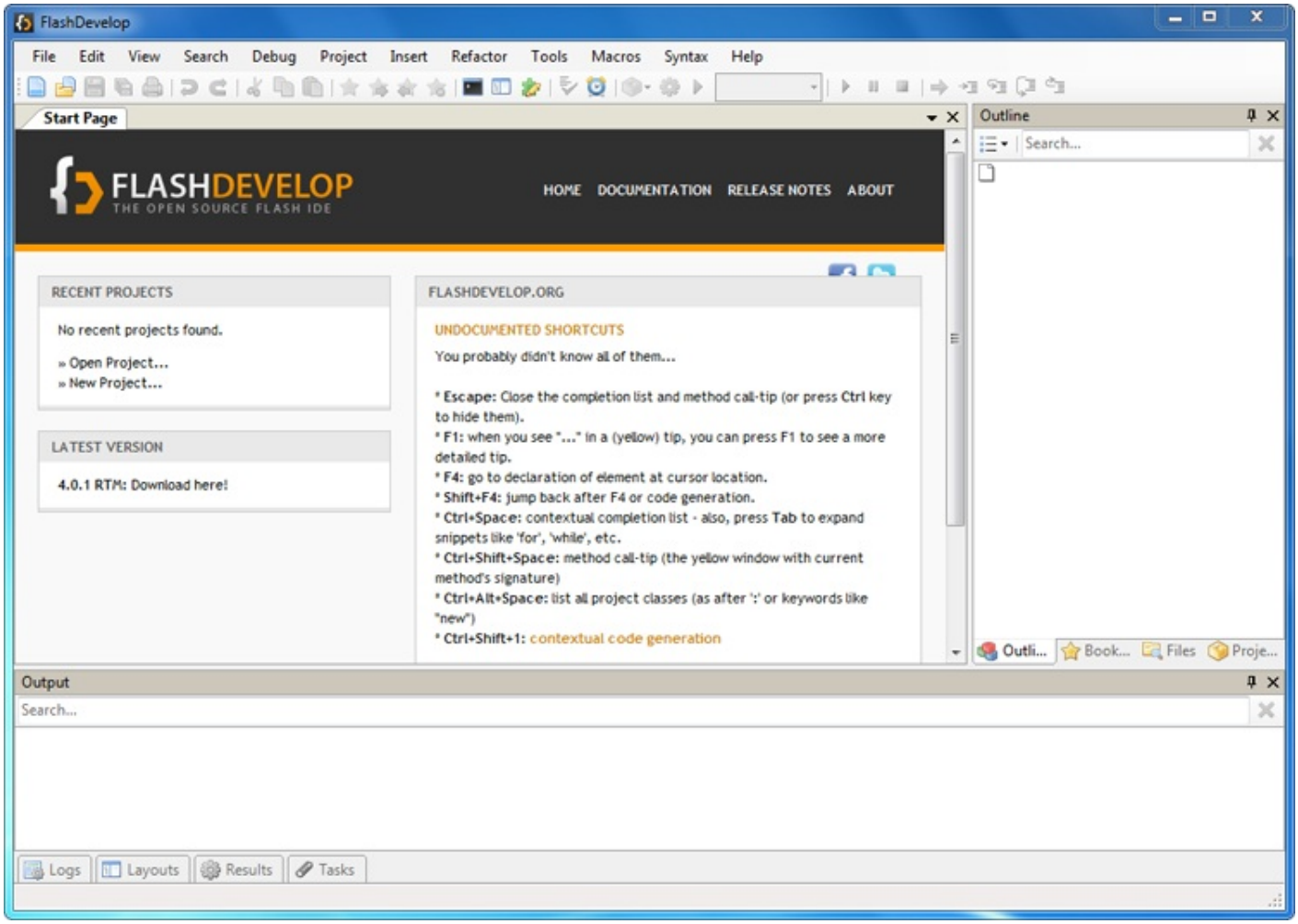
Cliquez sur `Finish` pour terminer l'installation.



Installation de FlashDevelop :

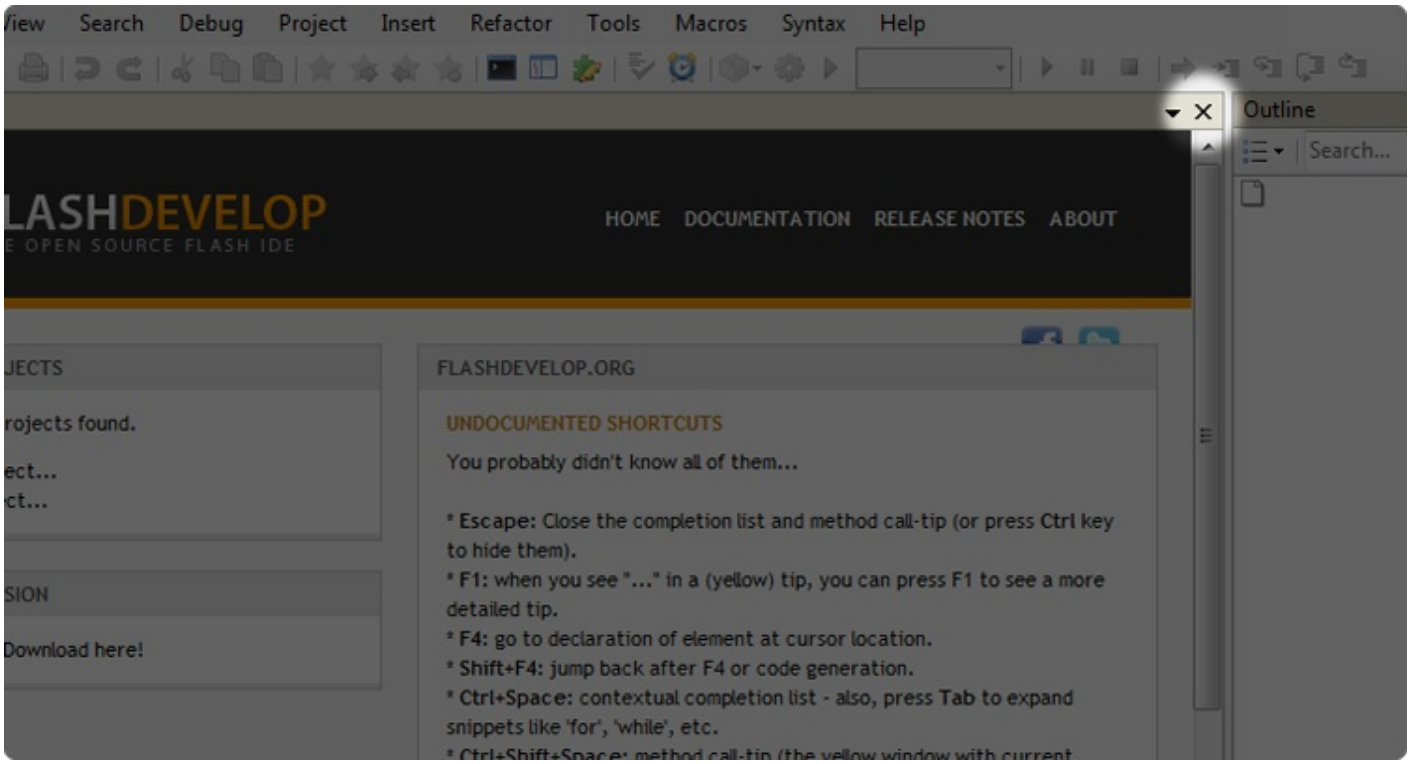
étape 5

Si Flashdevelop ne se lance pas tout seul, lancez-le. Vous arrivez sur l'écran d'accueil :



Page de démarrage de FlashDevelop

Vous pouvez fermer la page d'accueil en cliquant sur la croix correspondante :

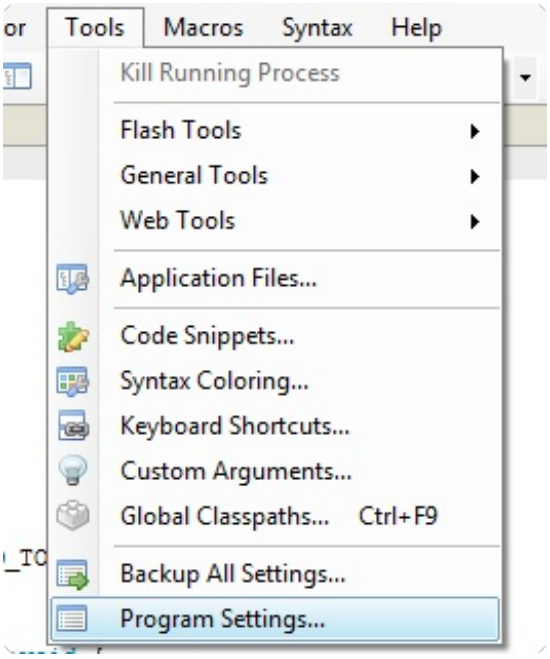


Fermeture de la page d'accueil

Un peu de paramétrage

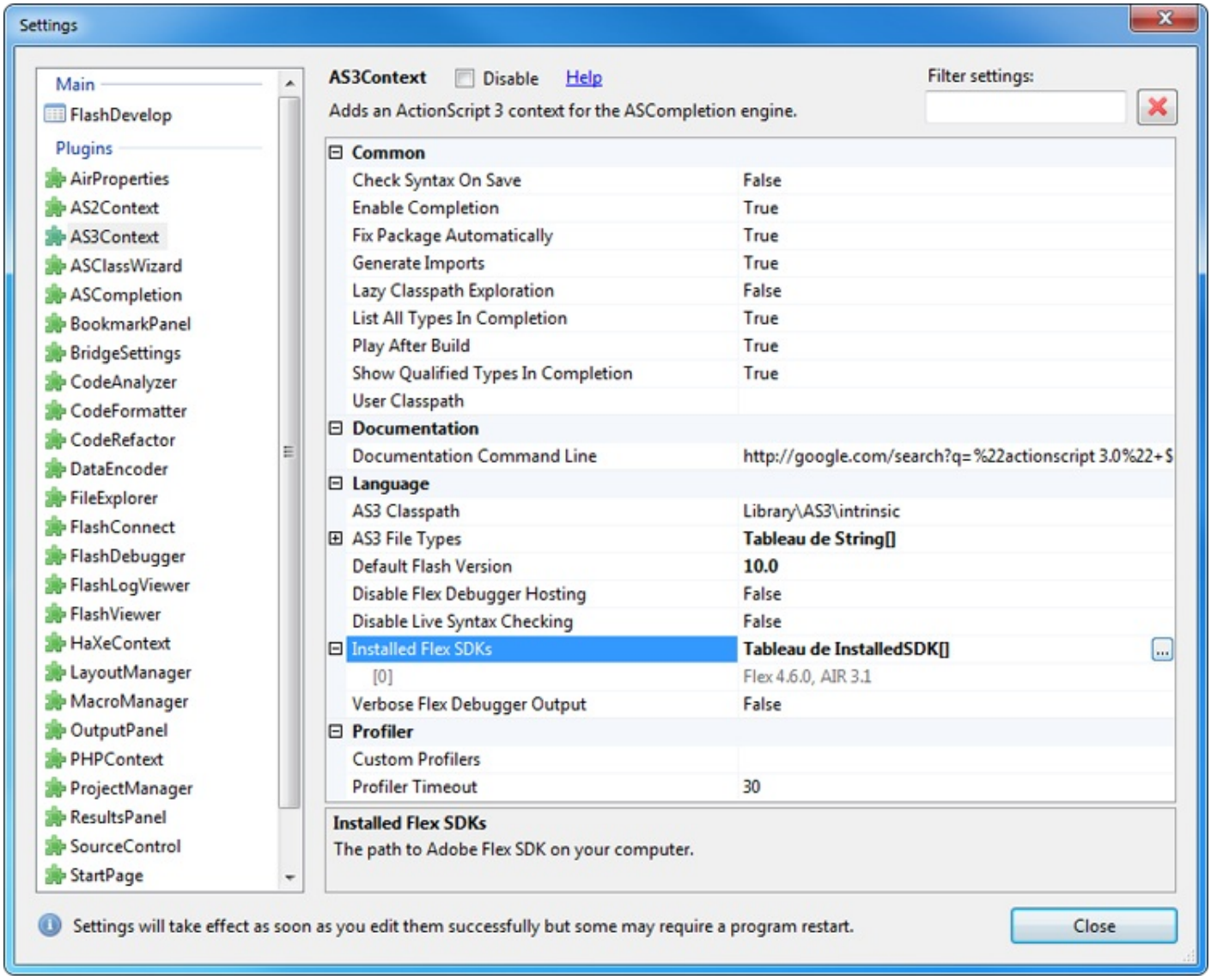
Si vous avez décoché `Install Flex SDK` lors de l'installation du logiciel, il faut lui dire où se trouve le *Flex SDK* que nous avions téléchargé au début avant de pouvoir l'utiliser : autrement, il ne pourra pas compiler notre projet, ce qui serait bien dommage. Ces manipulations vous seront également utiles si vous mettez à jour le *Flex SDK* plus tard.

Commençons par nous rendre dans les paramètres du logiciel, à l'aide du menu `Tools` :



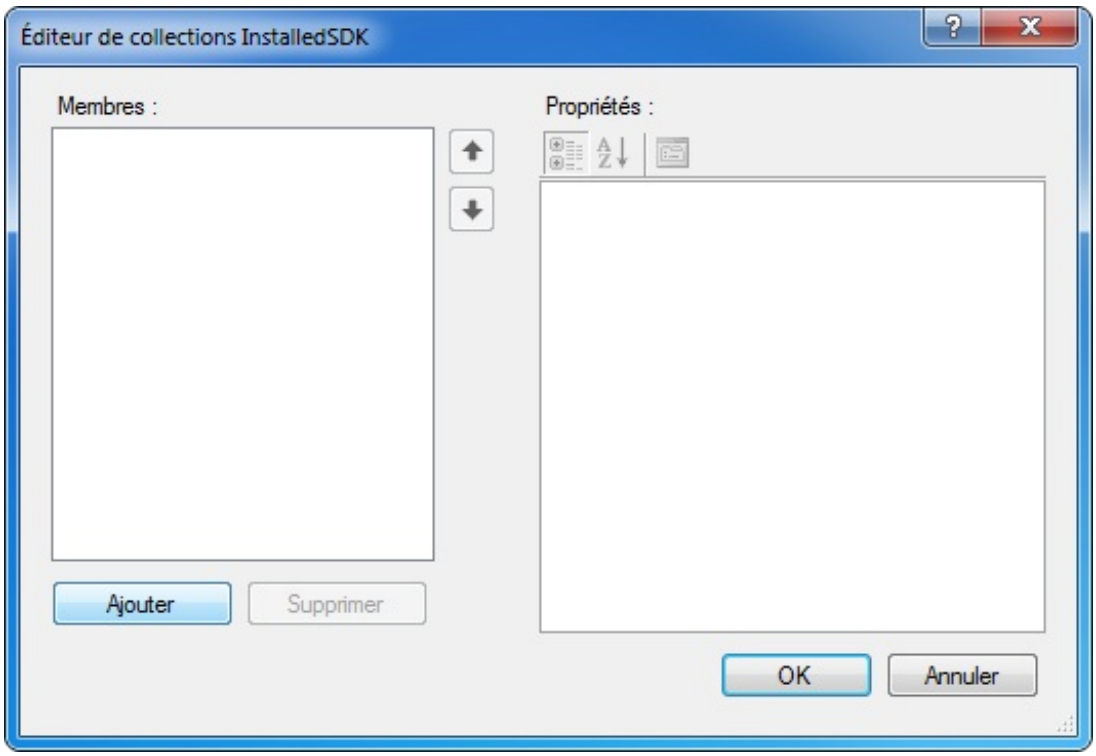
Accès aux paramètres de FlashDevelop

La fenêtre des paramètres de Flashdevelop s'ouvre ; sélectionnez `AS3Context` dans la liste de gauche, puis `Installed Flex SDKs` dans le panneau de droite. Ensuite, cliquez sur le petit bouton avec trois points :



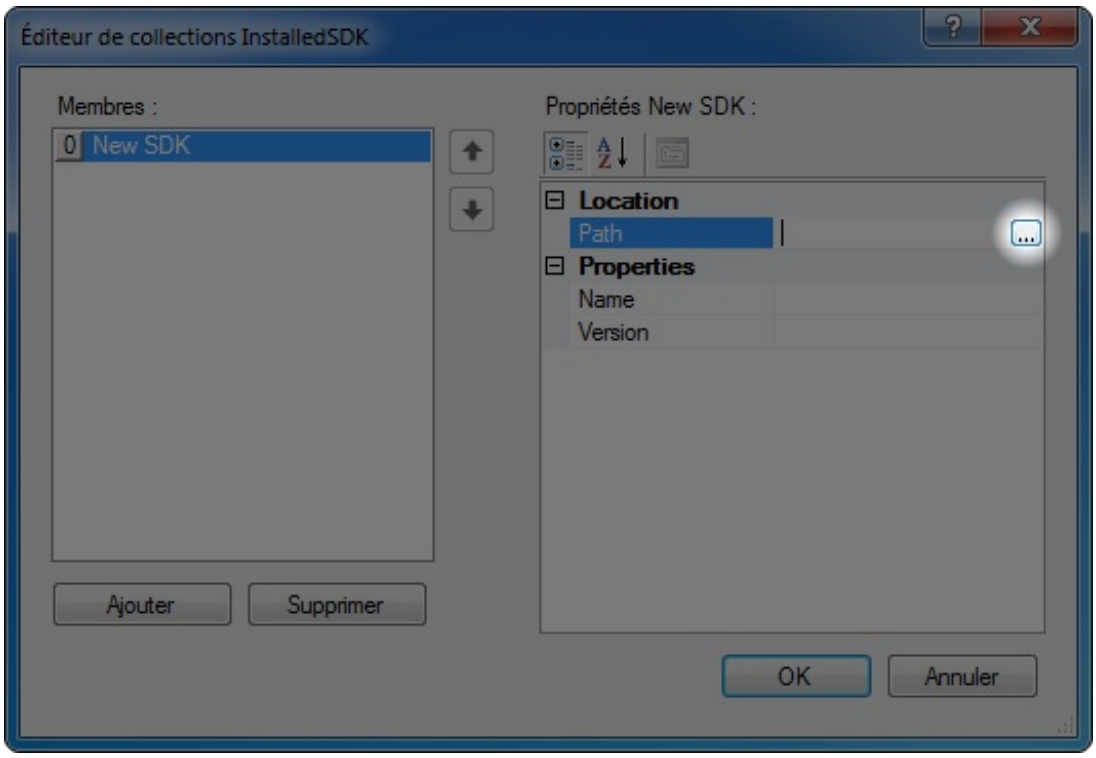
Paramètres de FlashDevelop

Le gestionnaire des SDK s'ouvre et est normalement vide. Nous allons ajouter notre SDK dans la liste : cliquez sur le bouton `Ajouter` en bas à gauche :



Éditeur de collections InstalledSDK

Un nouveau SDK est apparu dans la liste ! Il faut maintenant spécifier le chemin (*Path*) du SDK en le sélectionnant et en cliquant sur le petit bouton de droite à trois points :

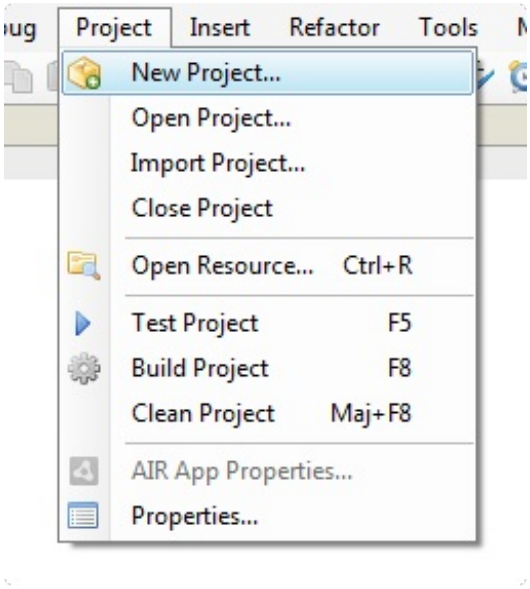


Mise en place du SDK

Choisissez le dossier où vous avez placé le SDK au début du chapitre, par exemple `D:\Flex SDK 4`. Pour finir, validez en cliquant sur le bouton `OK` et fermez la fenêtre des paramètres à l'aide du bouton `Close` situé en bas à droite.

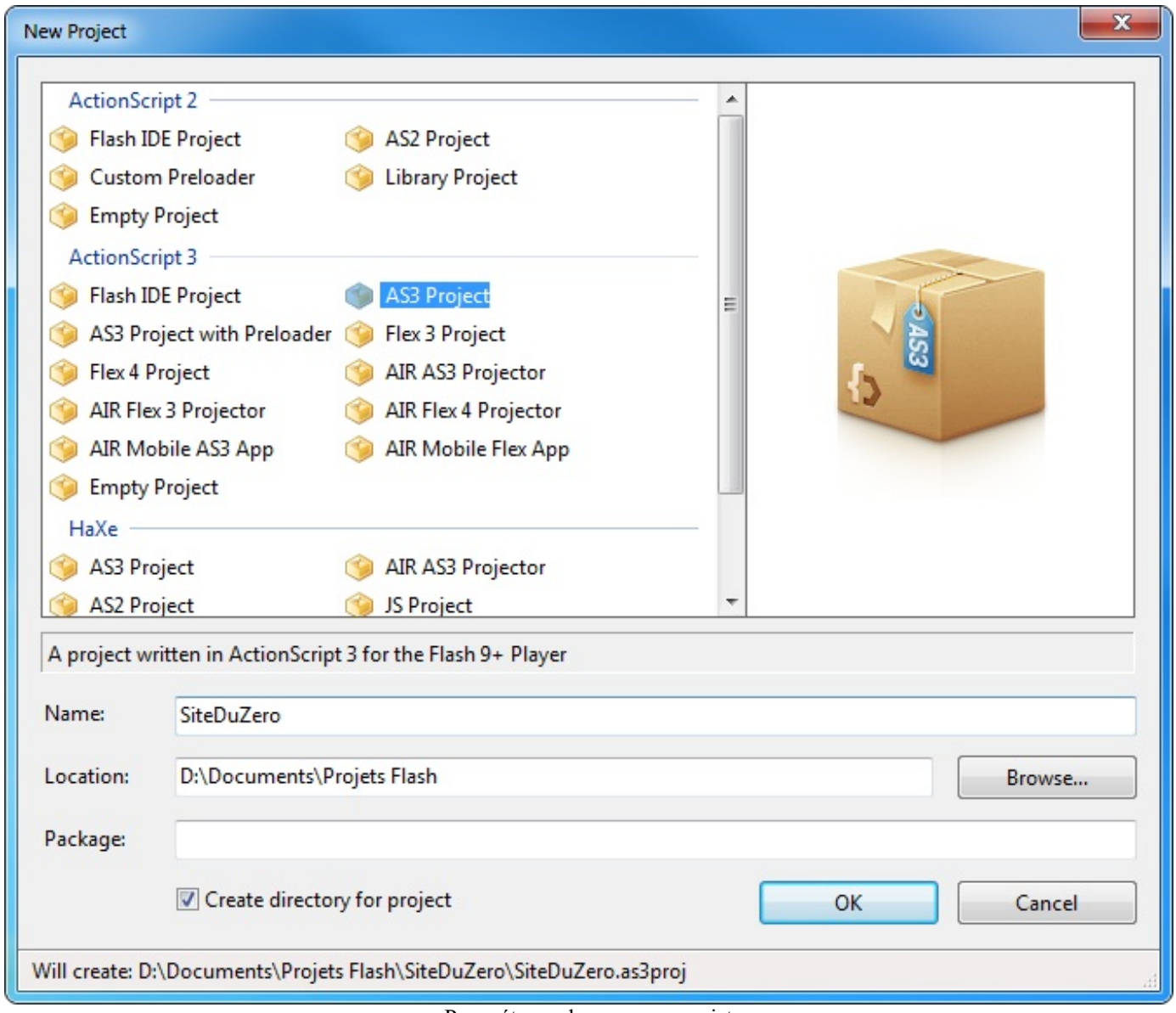
Créons un projet Actionscript

Pour pouvoir programmer sereinement au même endroit durant le cours, il nous faut créer un projet Actionscript 3. Pour cela, allez dans le menu `Project` de la barre de menus en haut, et cliquez sur `New Project`.



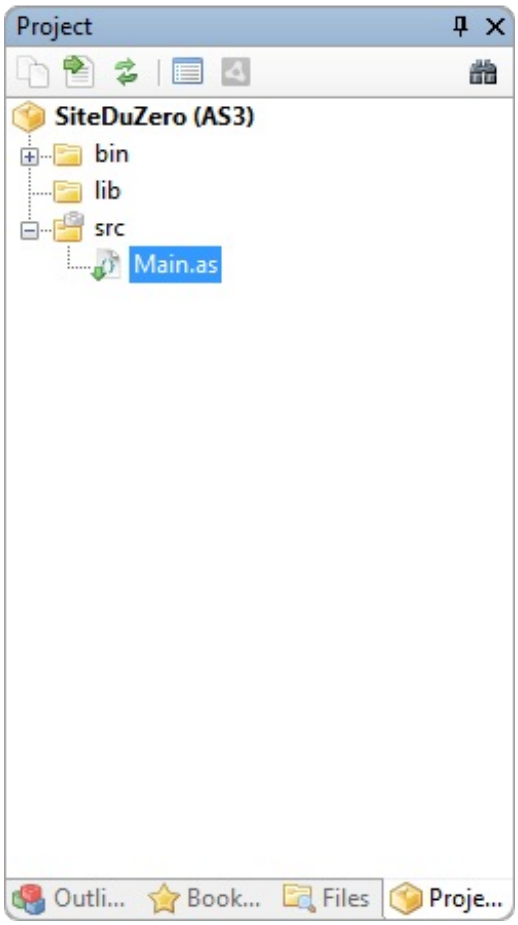
Création d'un nouveau projet

Une nouvelle fenêtre apparaît, proposant plusieurs types de projets. Sélectionnez `AS3 Project` et donnez un nom à votre projet dans le champ `Name`. Vous pouvez demander à Flashdevelop de vous créer automatiquement un dossier pour votre projet, en cochant la case `Create directory for project`.



Paramétrage du nouveau projet

Validez et notre projet est prêt ! Flashdevelop a créé pour nous les dossiers et fichiers de base qu'il faut pour commencer à travailler sur notre programme Flash.
Sélectionnez le panneau `Project` en bas à droite : il nous affiche l'arborescence de notre projet.



Arborescence du projet

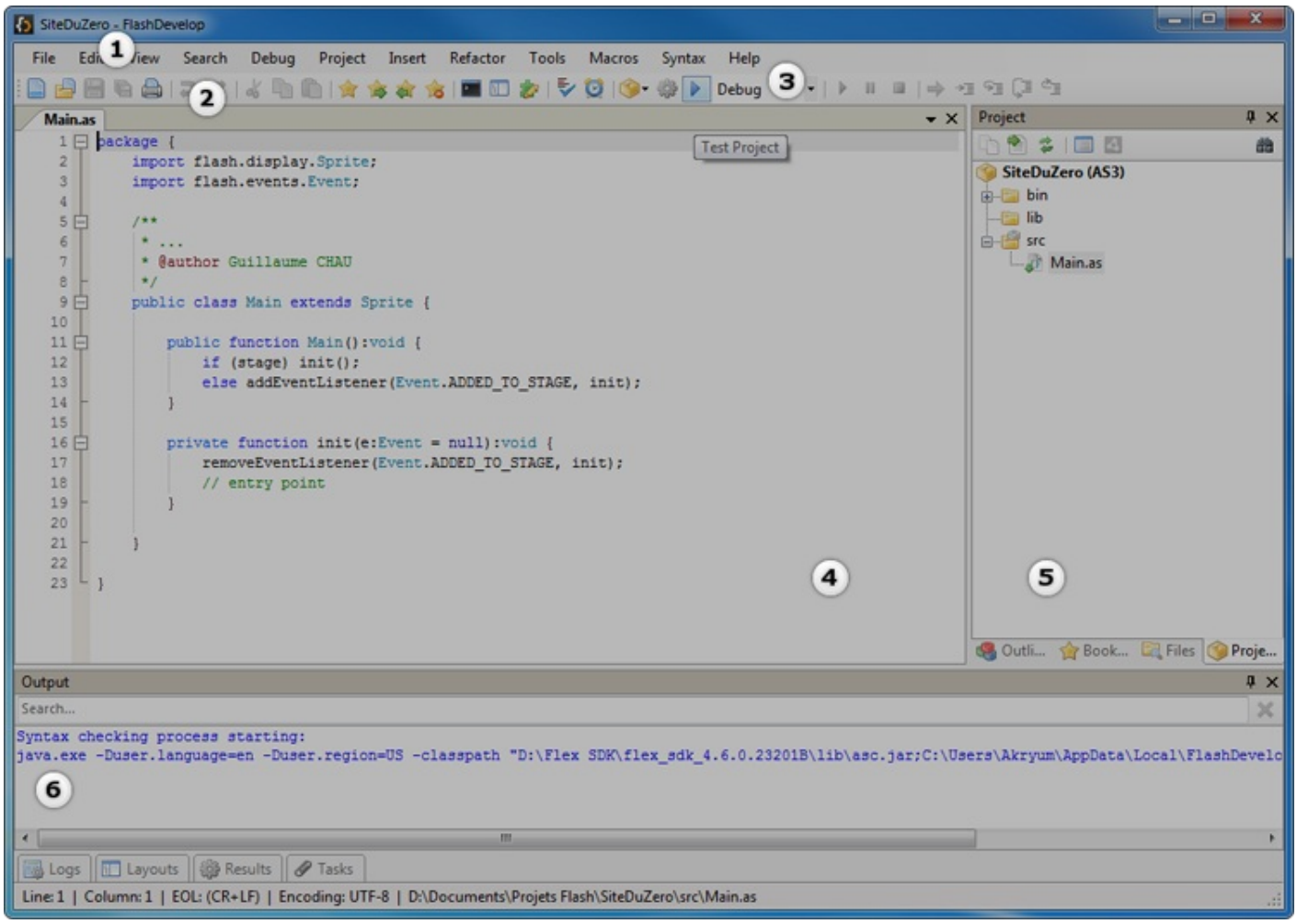
Le dossier `bin` doit contenir tous les médias qui seront chargés à l'exécution de notre programme (pour l'instant, nous allons le laisser tel qu'il est). C'est aussi le dossier où notre fichier SWF sera créé à la compilation.
Le dossier `lib` sert à regrouper tous les médias et librairies que vous pourriez importer dans votre programme, comme je l'ai expliqué au tout début du chapitre. Laissons-le vide également.
Enfin, le dossier `src` contient tous les fichiers de code qui composent notre programme. Étendez-le, et vous verrez que Flashdevelop a créé pour nous un fichier Actionscript principal (reconnaissable à la petite flèche verte), qu'il a nommé `Main.as`. Double-cliquez dessus pour l'ouvrir dans l'éditeur.

Bien utiliser Flashdevelop

Avoir de bons outils ne suffit pas : il est primordial de bien savoir les utiliser !
Ainsi, avant de tester notre projet, il est préférable de faire un tour du propriétaire.

L'interface

L'interface de Flashdevelop, bien qu'en anglais, est plutôt claire et surtout très pratique.
Détaillons-la ensemble.



L'interface de FlashDevelop

1. La barre de menu

Toutes les fonctionnalités de Flashdevelop sont accessibles dans ces menus.

- `File` (Fichier) : vous trouverez ici toutes les commandes en rapport avec les fichiers, comme la création de nouveaux documents, la sauvegarde, l'impression...
- `Edit` (Édition) : ce deuxième menu concerne le texte que vous tapez dans l'éditeur. Vous pouvez ainsi facilement annuler des modifications, copier-coller du texte, commenter du code sélectionné...
- `View` (Affichage) : ici, vous pouvez modifier la présentation de Flashdevelop, et ré-ouvrir les panneaux que vous auriez malencontreusement fermé.
- `Search` (Recherche) : si vous voulez effectuer des recherches de texte dans votre document ou dans le contenu des fichiers de votre projet, passez par ce menu !
- `Debug` (Débogage) : ce menu est un peu plus technique et concerne les sessions de débogage que vous effectuerez dans Flashdevelop (comme avec `fdb`). Nous en reparlerons plus tard.
- `Project` (Project) : tout ce qui touche à la gestion de vos projets, comme la création, l'ouverture, mais aussi pour tester votre projet actuel.
- `Insert` (Insertion) : ici vous pouvez insérer du texte spécial dans votre document, comme l'heure actuelle (`timestamp`) ou une couleur entre autres.
- `Refactor` (Refactorisation) : derrière ce terme barbare se cachent les opérations automatiques de maintenance et de mise en forme de votre code afin de le retravailler pour qu'il soit plus clair (`Code formatter`). Dans Flashdevelop se trouvent également des outils de génération de code pour travailler plus vite (`Code generator`).
- `Tools` (Outils) : ici vous trouverez des outils pour le développement en Flash, et plusieurs fenêtres de paramétrage du logiciel.
- `Macros` : il s'agit de scripts à lancer dans le logiciel pour automatiser certaines tâches ; toutefois nous n'aborderons pas les macros de Flashdevelop dans ce cours.
- `Syntax` (Syntaxe) : dans ce menu, vous pouvez spécifier à Flashdevelop dans quel langage vous êtes en train de coder, afin qu'il puisse colorier le code de manière adéquate. En général, on n'utilise pas ce menu, car Flashdevelop détecte le type de chaque fichier à l'aide de son extension.
- `Help` (Aide) : ce menu regroupe toutes les aides disponibles pour Flashdevelop, et permet également de vérifier les mises à jour du logiciel.

2. La barre de raccourcis

Les commandes les plus utilisées sont rassemblées ici pour pouvoir les utiliser directement, sans avoir à passer par les menus.

3. Les raccourcis de débogage

Cette zone de la barre des raccourcis est réservée aux boutons utiles pour compiler notre projet (`Build Project`), le tester (`Test Project`) et changer le mode de débogage (choisir `Debug` revient à ajouter le paramètre `-debug=true` au compilateur du *Flex SDK* comme nous l'avons vu plus haut).

4. L'éditeur de texte

C'est ici que vous écrirez vos programmes. À gauche sont affichés les numéros des lignes, et en haut se trouvent les différents documents ouverts sous forme d'onglets.

5. Les panneaux de développement

Ce sont des outils qui vous faciliteront la vie lorsque vous programmerez.

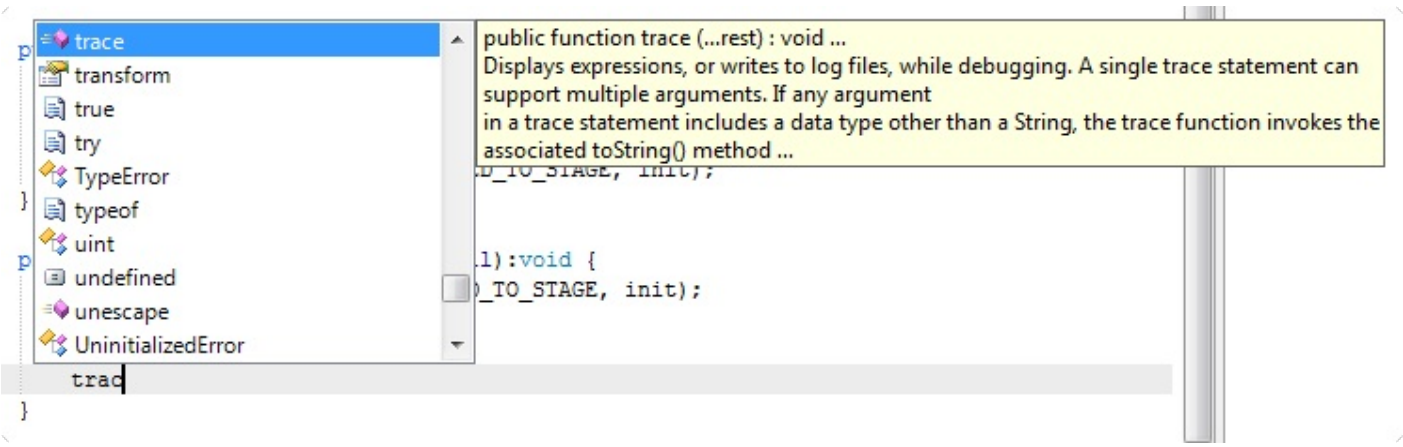
6. La console de sortie

Dans ce panneau seront affichés tous les messages de Flashdevelop (comme l'état de la compilation par exemple) et les messages du lecteur Flash (par exemple, avec la fonction `trace()`).

L'auto-complétion

Cette fonctionnalité est très utile et vous l'utiliserez en permanence pendant que vous programmerez : en effet, elle peut terminer ce que vous êtes en train de taper à votre place (toujours dans une optique de gain de temps), et peut également vous aider à choisir quoi taper grâce aux informations qu'elle propose.

Commençons par compléter notre classe `Main` : placez-vous dans la fonction `_init` après le commentaire `// entry point` (point d'entrée) et commencez à taper `trace`. Surprise ! Un menu s'ouvre au bout du quatrième caractère, vous proposant d'ajouter la fonction `trace()`, avec en bonus sa description !



Auto-complétion de la fonction trace()

Pour valider votre choix, appuyez sur Entrée, et le mot `trace` est automatiquement terminé ! Bien entendu, cela n'a pas une grande utilité dans notre cas, car le nom de la fonction est très court. Mais imaginez si vous aviez à retenir toutes les fonctions, et en plus si leur nom fait vingt caractères !

Il est également possible d'activer l'auto-complétion à tout moment, même si l'on n'a rien encore tapé : il suffit d'appuyer simultanément sur les touches Ctrl et Espace.

Compiler et tester notre projet

Terminez votre ligne de code pour que votre programme affiche « Hello world ! » dans la console. Comment ça, vous ne vous souvenez plus comment faire ? 
Bon d'accord, je vous donne la ligne à ajouter, mais c'est bien parce que c'est vous :

Code : Actionscript

```
trace("Hello world !");
```

Voici à quoi doit ressembler le fichier après modifications :

Code : Actionscript

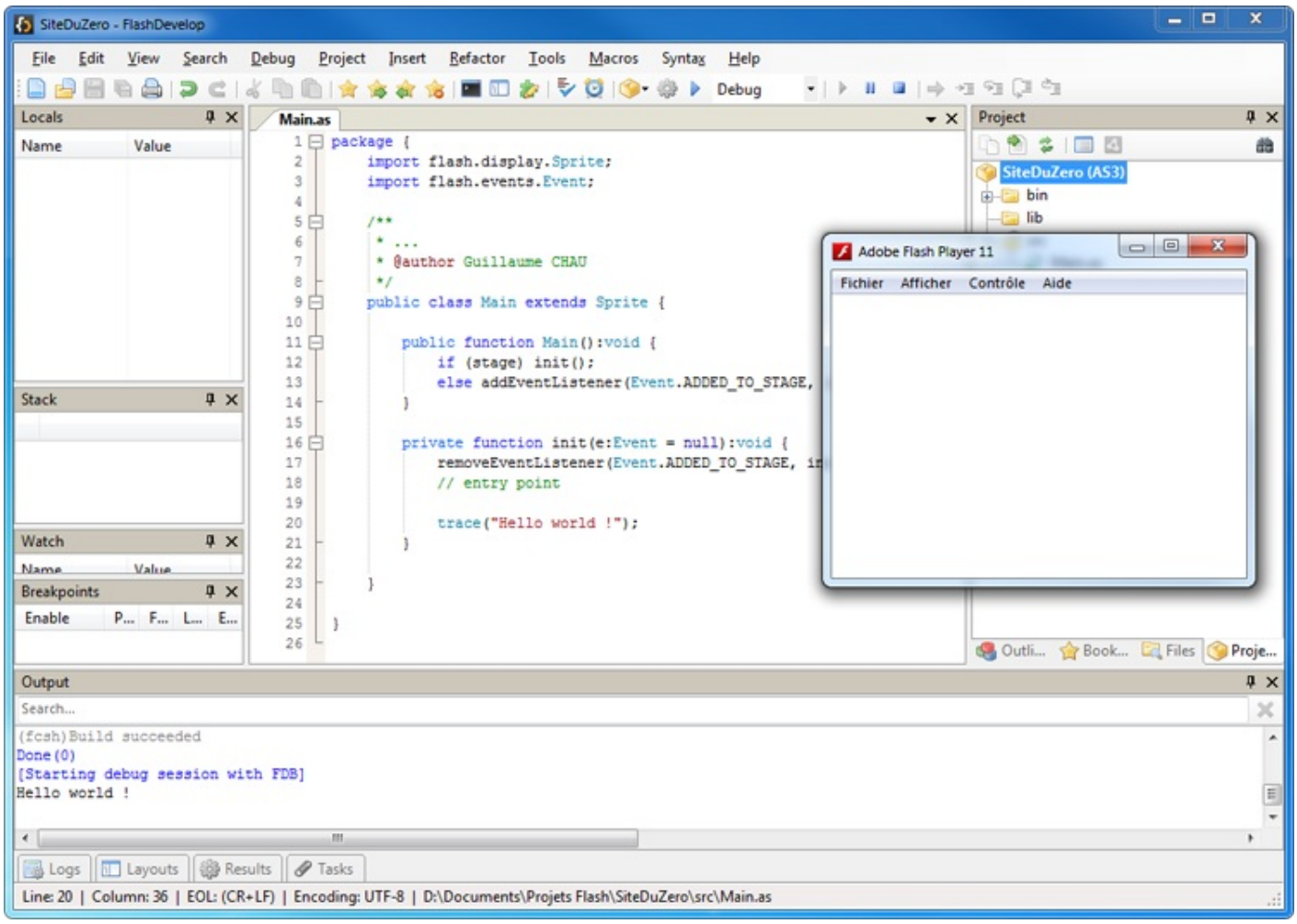
```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    /**
* ...
* @author Guillaume
*/
    public class Main extends Sprite {

        public function Main():void {
            if (stage)
                init();
            else
                addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // entry point

            trace("Hello world !");
        }

    }

}
```

 Mais, pourquoi le code est différent par rapport à tout à l'heure ?

Très bonne question ! Flashdevelop, en créant notre projet, a automatiquement rempli cette classe avec le code de base de telle sorte qu'il n'y ait aucun problème pour nous. Les fonctions qu'il a rajouté sont utiles dans certains cas pour éviter des erreurs d'affichage. Retenez qu'il faut commencer à programmer juste après le commentaire `// entry point` seulement dans notre classe `Main`. Dans les autres classes que vous créerez, ce code supplémentaire ne sera pas nécessaire.

Une fois que votre classe `Main` est prête, lancez la compilation en cliquant sur le bouton en forme de flèche bleue (`Test Project`) dans la barre des raccourcis ou en appuyant simultanément sur les touches Ctrl et Entrée. Vous remarquerez qu'il y a un bouton `Build Project` immédiatement à gauche, qui ne sert qu'à compiler notre projet sans le lancer.

Si tout ce passe bien, une fenêtre du lecteur Flash s'ouvre avec votre programme, et la console affiche notre message « Hello world ! » :



Lancement du programme

C'est aussi simple que cela !

En résumé

- L'opération de vérification des erreurs, d'assemblage des différents fichiers et de compression d'un projet s'appelle la **compilation**.
- Le Flex SDK est un ensemble d'outils de développement, et notamment un compilateur permettant d'aboutir au fichier `swf`.
- FlashDevelop est un IDE, c'est-à-dire un outil qui permet d'écrire, de compiler et de tester du code Actionscript plus facilement.
- En Actionscript, le code s'écrit à l'intérieur de **fonctions**, elles-mêmes contenues dans une **classe**, le tout placé dans un **package**.
- Il est possible d'insérer des commentaires au milieu du code grâce aux caractères « / / », « / * » et « * / ».
- Il existe une fonction `trace()` qui permet d'écrire du texte dans la console, et est principalement utilisée pour les étapes de débogage.

Les variables

Les variables sont un des éléments les plus importants dans tous les langages de programmation : elles permettent de *mémoriser* des informations de toutes sortes. Sans elles, les programmes seraient tous très basiques et moins puissants, le monde informatique serait alors d'un ennui mortel. 

Un petit exemple concret : vous voulez que votre programme vous dise bonjour, après lui avoir donné votre nom. Une variable (appelée `monNom` par exemple) se chargera alors de mémoriser le mot que vous lui entrerez pour plus tard permettre au programme de vous appeler par votre nom : il lui suffira de regarder ce qu'il se trouve dans la variable `monNom`.

Déclarer et utiliser des variables

Déclaration

Le mot-clé `var`

La première chose à faire avant d'utiliser des variables, c'est de les *créer* :

Code : Actionscript

```
var maVariable;
```

Pour cela, nous avons à notre disposition un mot-clé (ou mot réservé) obligatoire : **`var`**. Cette instruction permet de déclarer une variable ; elle s'utilise de la façon suivante : `var` suivit d'un espace et du nom de la variable.

Petit rappel : un mot-clé (ou réservé) est une expression qui est utilisée par le langage et qui lui est donc réservée, c'est-à-dire que vous ne pouvez pas appeler une variable `var` ou utiliser ce mot réservé pour autre chose que son utilité première.

Désormais, vous savez créer une variable, mais le code précédent n'est pas très utile. En effet, chaque variable possède un **type**, indispensable, qui décrit son comportement et son utilité (par exemple : nombre, entier, chaîne de caractères...). Sans ce type, le langage ne saurait pas à qui il aurait à faire en lisant une variable, et surtout *comment* l'utiliser.

N'oubliez pas de mettre le point-virgule en fin de ligne !

Comme pour les noms de classes, il est préférable d'utiliser la notation Camel pour le nom de nos variables : `leNomDeMaVariable` au lieu de `lenomdemavariable`.
Vous pouvez également utiliser l'underscore (_) pour simuler les espaces qui eux sont interdits.

Le typage

Il faut donc assigner un type à chaque nouvelle variable, pour que le programme puisse la reconnaître. Pour cela, on utilise les deux points ainsi :

Code : Actionscript

```
var maVariable:sonType;
```

Ce type suivra la variable tout au long de l'exécution du programme.

Avant, en Actionscript 2, il était possible de déclarer des variables sans les typer, le lecteur flash se chargeait de deviner le type de ces variables. Bien entendu, cette technique est à proscrire pour des raisons de performances et de rigueur (imaginez plusieurs centaines de variables à qui vous avez affaire, mais dont vous ne connaissez pas le type). Désormais, les variables doivent être typées à la création : on appelle cela le typage strict. C'est une des principales raisons de l'amélioration des performances des animations depuis le passage à l'Actionscript 3.

Différents types que vous pourrez utiliser seront détaillés un peu plus loin.

Initialiser la nouvelle variable

Après avoir créé une variable, il vous prendra peut-être l'envie d'y mettre quelque chose pour le réutiliser plus tard :

Code : Actionscript

```
var maVariable:sonType = ceQueJeMetsDedans;
```

Vous remarquerez qu'on peut remplir une variable directement à sa création (ce qui est drôlement pratique), en mettant un signe égal après le type ; on appelle cette opération l'**initialisation**. Ainsi, dès qu'elle sera créée, la variable `maVariable`sera du type `sonType`et contiendra `ceQueJeMetsDedans`.

Vous n'êtes évidemment pas obligés d'initialiser les variables, vous pourrez leur donner une valeur (ou un contenu si vous préférez) plus tard à l'aide de l'affectation.

Les valeurs par défaut

Lorsque vous créez une variable et que vous ne l'initialisez pas, elle n'est pas tout à fait vide ; en effet, elle contient automatiquement une *valeur par défaut* qui sera souvent **`null`**, sauf quelques cas particuliers dépendant du type de la variable. Ainsi, si vous écrivez ceci :

Code : Actionscript

```
var maVariable:sonType;
```

`maVariable` contiendra sa valeur par défaut, donnée par son type.

Le mot-clé **`null`** est réservé à l'Actionscript, vous ne pouvez donc pas appeler une variable `null`. Cette valeur remplace l'ancien mot-clé équivalent **`undefined`** valable en Actionscript 2. Néanmoins, une variable que vous n'initialisez pas contient **`undefined`** si elle n'a pas de type, et vous pourrez rencontrer ce mot-clé avec les *tableaux* dans un des chapitres suivants.

Utiliser les variables

Affectation

Tout au long de l'exécution du programme, vous aurez sûrement besoin de modifier le contenu d'une variable en fonction des besoins du moment. La méthode est presque identique à l'initialisation, car on utilise à nouveau le signe égal :

Code : Actionscript

```
maVariable = nouveauContenu;
```

Ainsi, la valeur `nouveauContenu` sera stockée dans la variable `maVariable`.

Il est absolument interdit d'affecter une valeur à une variable si cette dernière n'existe pas. Cela n'aurait aucun sens, et le compilateur vous le fera savoir en refusant de compiler votre programme. N'oubliez donc pas de déclarer vos variables avant de les utiliser.

Lecture

Il est tout aussi intéressant de pouvoir lire une variable pour utiliser son contenu : par exemple, le programme aura besoin de regarder ce que contient la variable `monNom` pour pouvoir afficher « Bonjour Georges ! » si j'ai mis « Georges » dans la variable avant.
Par exemple, pour copier le contenu de la variable `a` dans la variable `b`, il faut donc écrire :

Code : Actionscript

```
b = a; // Je prends ce qu'il y a dans la variable a, et je le mets
dans b
```

Vous pouvez voir ainsi que la façon de procéder est très simple : il suffit de renseigner le nom de la variable.

Petite piqûre de rappel : `// Je prends ce qu'il y a dans la variable a, et je le mets dans b` est un *commentaire*; il permet d'écrire des informations sur le programme à destination d'éventuels lecteurs ou pour vous-mêmes, afin de vous rappeler à quoi sert ce que vous avez tapé là, par exemple. Les commentaires n'influent en aucune façon sur le fonctionnement du programme, ils sont tout simplement ignorés.

Avec les déclarations des variables, cela donnerait :

Code : Actionscript

```
var a:typeDeA = contenuA;
var b:typeDeB = contenuB;
b = a; // Je prends ce qu'il y a dans la variable a, et je le mets
dans b
// La variable b contient maintenant "contenuA"
```

Les nombres
Les différents types

Comme vous pouvez vous en douter, les nombres sont très utilisés dans le monde de l'informatique, étant donné que le *numérique* est basé sur des suites de 0 et de 1. Il existe différents types de nombres en ActionScript 3, ayant chacun leurs spécificités.

Le type `int`

Le type `int` sert à manipuler des entiers relatifs. Voici un exemple de déclaration et d'initialisation de ce type de variables :

Code : Actionscript

```
var monEntier:int = -100;
```

Ce type permet de manipuler des nombres codés sur 32 bits (c'est-à-dire 32 « 0 » ou « 1 »), donc compris entre -2 147 483 648 et 2 147 483 647. *Si vous sortez de cet encadrement, vous obtiendrez une erreur.* La valeur par défaut de ce type est `0`.

Pour accéder rapidement à ces deux valeurs, utilisez respectivement `int.MIN_VALUE` et `int.MAX_VALUE`. Ce sont des variables un peu spéciales, car elles sont utilisables partout dans votre code et on ne peut que les lire. On les appelle des **constantes**, notion que nous aborderons dans le chapitre sur l'orienté objet.

Le type `uint`

Le type `uint` sert à manipuler des entiers naturels ou *non-signés* (c'est-à-dire positifs), voici un exemple de déclaration et d'initialisation :

Code : Actionscript

```
var monEntier:uint = 42;
```

Le type `uint` permet d'utiliser des entiers naturels codés sur 32 bits également, donc compris entre 0 et 4 294 967 295. Comme le type `int`, la valeur par défaut est `0`, et vous obtiendrez aussi une erreur si vous sortez de cet encadrement.
Dans certaines situations, l'utilisation de ce type ralentit légèrement l'exécution de votre programme. Ainsi, je vous conseille d'utiliser le type `int` si les nombres que vous voulez manipuler sont inférieurs à 2 147 483 647.

Pour accéder rapidement à ces deux valeurs, utilisez respectivement `uint.MIN_VALUE` et `uint.MAX_VALUE`. Elles sont également utilisables en lecture seule partout dans votre code.

Le type ***Number***

Le type `Number` sert à manipuler tous les nombres (entiers comme flottants), dans un intervalle extrêmement grand. On peut presque considérer qu'on peut y mettre tous les nombres.
Voici comment les utiliser :

Code : Actionscript

```
var monNombre:Number = 3.1415;
```

Comme dans la plupart des langages de programmation, et de façon générale en informatique, on utilise la notation anglaise des nombres flottants (ou à virgule). C'est-à-dire qu'à la place de la virgule, on met un point : 3,14 écrit en français donne `3.14` en Actionscript.

La valeur par défaut de ce type est **`NaN`**.

Le mot-clé **`NaN`** signifie *Not a Number* (pas un nombre) : votre nombre prend cette valeur si vous ne l'initialisez pas à une certaine valeur, ou si vous tentez d'y stocker autre chose qu'un nombre.

Ainsi, le code suivant ne ferait pas d'erreur, mais la variable `nbr` aurait pour valeur **`NaN`** :

Code : Actionscript

```
var nbr:Number = Number("Je veux un nombre !"); // On force la
variable à contenir du texte...
trace(nbr); // Affiche : NaN
```

Pour accéder rapidement à la valeur minimum ou à la valeur maximum autorisée, utilisez respectivement `Number.MIN_VALUE` et `Number.MAX_VALUE`. Une nouvelle fois, elles sont utilisables en lecture seule partout dans votre code.

Vous êtes curieux de savoir quels sont le minimum et le maximum autorisés ? Voici le code pour les afficher :

Code : Actionscript

```
trace(Number.MIN_VALUE + " à " + Number.MAX_VALUE);
// Affiche : 4.940656458412465e-324 à 1.79769313486231e+308
// Le "e" signifie "fois dix puissance" ; par exemple, 1e+10
équivaut à 1x10^10 = 10 000 000 000
// 1.79769313486231e+308 est donc un nombre à 309 chiffres :p
```

Si par mégarde vous essayez d'affecter un nombre flottant (c'est-à-dire à virgule, comme 3.14) à une variable de type

`int` ou `uint`, il sera automatiquement arrondi à l'entier inférieur avant d'être stocké dans la variable. Par exemple, 3.14 deviendra 3, et 45.9 deviendra 45.

Opérations sur les nombres

Les opérateurs de base

Pour effectuer une opération entre deux nombres, on procède comme sur les cahiers de Maths à l'école ! 
L'opération est effectuée lors de l'exécution du programme et le résultat peut être stocké dans une variable `monNombre` par exemple. Voici un tableau qui regroupe les opérations de base :

Nom de l'opération	Symbole	Exemple
Addition	+	`monNombre = 1 + 4; // monNombre = 5`
Soustraction	-	`monNombre = 8 - 3; // monNombre = 5`
Multiplication	*	`monNombre = 2 * 3; // monNombre = 6`
Division	/	`monNombre = 8 / 4; // monNombre = 2`
Modulo	%	`monNombre = 8 % 5; // monNombre = 3`

Pour ceux qui ne le connaîtraient pas, le modulo est un opérateur moins courant qui permet de renvoyer le reste de la division euclidienne entre deux nombres.

Ces opérations peuvent être effectuées sur des variables des trois types de nombres que nous avons vu précédemment, même en les mélangeant. Voici quelques exemples de calculs :

Code : Actionscript

```
var unEntier:uint = 3 + 42;
var unAutreEntier:int = -25;
var monResultat:Number = unEntier * unAutreEntier;
monResultat = monResultat / 100;
trace(monResultat); // Affiche : -11.25
```

Contrairement à beaucoup d'autres langages, diviser par zéro ne fera pas planter votre programme... Le résultat de l'opération sera en fait `Number.POSITIVE_INFINITY`, autrement dit, le nombre infini ! Faites très attention de vérifier qu'une telle chose arrive, sinon vous pourriez avoir des surprises lors de l'exécution de votre programme...

Notez également qu'il faut être prudent sur le type de variables utilisé pour les calculs. Je rappelle qu'un nombre à virgule sera automatiquement arrondi à l'entier inférieur si vous tentez de l'affecter à une variable de type `int` ou `uint`. Repérez donc ce qui se déroule au fil de ces instructions :

Code : Actionscript

```
var unEntier:uint = 2;
var unNombre:Number = 3.14;
var monResultat:int = unEntier + unNombre;
trace(monResultat); // Affiche : 5
```

Simplifier les calculs

Comme dans beaucoup de langages, il est possible en Actionscript de simplifier des calculs de ce genre :

Code : Actionscript

```
monResultat = monResultat / 100;
```

Ainsi l'écriture de cette instruction peut être simplifiée et réduite sous la forme :

Code : Actionscript

```
monResultat /= 100;
```

Ce code est donc plus rapide à écrire, et provoque le même résultat que précédemment. Bien évidemment cette manipulation n'est pas réservée à la division, mais peut être effectuée avec n'importe quel autre opérateur arithmétique : +=, -=, *=, /= et %=.

Nous avons à présent fait le tour des opérateurs disponibles en Actionscript.

Mais qu'en est-il des autres opérations mathématiques plus complexes, comme la racine carrée ?

En effet, il n'existe pas d'opérateurs arithmétiques en Actionscript 3 pour effectuer de telles opérations. Heureusement, une classe un peu spéciale appelée `Math` est fournie par Flash.

La classe `Math`

Cette classe n'a pas besoin d'être importée, elle est accessible en permanence. Elle contient une flopée d'outils mathématiques très utiles, comme les puissances, les fonctions trigonométriques, les nombres aléatoires...

Les puissances

Ces fonctions de la classe `Math` vous permettent de manipuler les puissances sur des nombres :

Code : Actionscript

```
var monNombre:Number = 42;
// Elever à la puissance
trace(Math.pow(monNombre, 5));
// Racine carrée
trace(Math.sqrt(monNombre));
```

Les arrondis

Il existe trois types d'arrondis : l'arrondi classique, l'arrondi à l'entier inférieur le plus proche et l'arrondi à l'entier supérieur le plus proche :

Code : Actionscript

```
var monNombre:Number = 3.1415;
// Arrondi
trace(Math.round(monNombre)); // 3
// Entier inférieur
trace(Math.floor(monNombre)); // 3
// Entier supérieur
trace(Math.ceil(monNombre)); // 4
```

Trigonométrie

Cosinus, sinus, tangente, arc-cosinus, arc-sinus et arc-tangente sont des fonctions trigonométriques que nous propose la classe `Math` :

La valeur de π est accessible à l'aide de `Math.PI`.

Code : Actionscript

```
var angle1:Number = Math.PI / 2;
var angle2:Number = Math.PI / 6;

// Cosinus
trace(Math.cos(angle1));
// 6.123233995736766e-17

// Sinus
trace(Math.sin(angle1));
// 1

// Tangente
trace(Math.tan(angle1));
// 16331239353195370

// ArcCosinus
trace(Math.acos(angle2 / angle1));
// 1.2309594173407747

// ArcSinus
trace(Math.asin(angle2 / angle1));
// 0.3398369094541219

// ArcTangente
trace(Math.atan(angle1));
// 1.0038848218538872
```

Les angles sont toujours exprimés en radians.

Il existe une variante de la fonction arc-tangente en Actionscript 3 : `Math.atan2()`. Elle sert principalement à calculer sans erreur l'angle entre deux positions. Nous en aurons besoin plus loin dans le cours.

Nombre aléatoire

Il serait très intéressant de fabriquer des nombres aléatoires, pour des jeux par exemple. Cela est possible avec la fonction `Math.random()` :

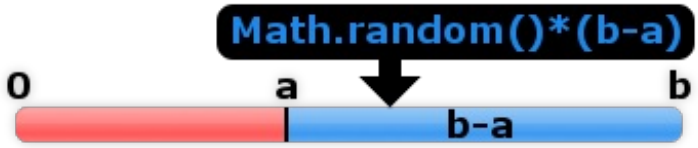
Code : Actionscript

```
trace(Math.random()); // Affiche un nombre flottant aléatoire
compris entre 0 et 1
```

Pour générer un nombre aléatoire entre deux valeurs `a` et `b`, il faut utiliser la formule suivante :

Code : Actionscript

```
trace(a + Math.random() * (b - a));
```



Création d'un nombre aléatoire

Les chaînes de caractères

Les chaînes de caractères sont également très utilisées : il s'agit d'une suite de caractères qui forme du texte. Par exemple, « Hello world ! » est une chaîne de caractères ; son premier caractère est le « H », et son dernier caractère est le « ! ».

Une chaîne de caractères est toujours entourée de guillemets, comme nous l'avons vu dans le chapitre précédent :

Code : Actionscript

```
trace("Hello world !");
```

Mais vous pouvez aussi mettre des apostrophes à la place des guillemets :

Code : Actionscript

```
trace('Hello world !');
```


Attention toutefois à ne pas mélanger les deux, cela ne marcherait pas. Ainsi, le code `trace("Hello world !');` est incorrect.

Échappement des caractères spéciaux


Mais si je veux mettre des guillemets ou des apostrophes dans ma chaîne de caractères ?

Je vous voyais venir ! Effectivement, mettre des guillemets dans une chaîne de caractères à guillemets ou des apostrophes dans une chaîne de caractères à apostrophes serait problématique : en effet, le compilateur pensera que vous avez terminé votre chaîne au deuxième guillemet ou apostrophe rencontré, et se demandera pourquoi diable d'autres caractères se baladent derrière ! 

Dans cet exemple, vous pouvez voir que la coloration syntaxique nous montre le problème :

Code : Actionscript

```
trace("Hello.swf a dit : "Hello world !"");
```

En effet, le « Hello world ! » n'est dans ce cas plus considéré comme faisant parti de la chaîne de caractères...

Pour remédier à cela, il faut *échapper* le ou les caractères qui posent problème. Cela consiste à mettre un autre caractère spécial, l'antislash (\), qui permettra de dire que le caractère suivant doit être pris pour un caractère tout à fait banal dans notre chaîne.

Ainsi, le code correct serait :

Code : Actionscript

```
trace("Hello.swf a dit : \"Hello world !\"");
```

Vous pouvez également remplacer les guillemets par des apostrophes dans les cas où ça vous arrange de faire ainsi :

Code : Actionscript

```
trace('Hello.swf a dit : "Hello world !"');
```

Plus besoin d'échapper, car ce n'est plus le caractère guillemet qui précise où débute et où se termine la chaîne, mais l'apostrophe. Par contre, si vous voulez mettre une apostrophe en plus, le problème va revenir :

Code : Actionscript

```
trace('Hello.swf m'a dit : "Hello world !"');
```

Encore une fois, le compilateur ne va vraiment rien comprendre à ce que vous lui écrivez. Solution : échapper le caractère apostrophe qui pose problème ! 

Code : Actionscript

```
trace('Hello.swf m\'a dit : "Hello world !"');
```

Vous pouvez systématiquement échapper les guillemets et apostrophes dans les chaînes de caractères si cela vous met à l'aise. 

Code : Actionscript

```
trace('Hello.swf m\'a dit : \"Hello world !\"');
```

Utiliser les variables

Maintenant, si nous voulons mémoriser des chaînes de caractères, il va falloir les ranger dans des variables de type `String`. Par exemple, pour mémoriser notre phrase « Hello world ! » dans la variable `coucou`, il faut procéder ainsi :

Code : Actionscript

```
var coucou:String = "Hello world !";
trace(coucou); // Affiche : Hello world !
```

La valeur par défaut d'une variable de type `String` est **`null`**.

Concaténation de chaînes

Concaténer deux chaînes de caractères consiste à les assembler pour ne former qu'une seule chaîne. Par exemple, concaténer la chaîne « Hello » avec la chaîne « world ! » donnerait une nouvelle chaîne plus grande : « Hello world ! ».
Pour cela, il faut utiliser l'opérateur + (qui sert aussi à additionner deux nombres), de cette façon :

Code : Actionscript

```
"Hello" + " world !"
```

Ainsi, si l'on voulait afficher notre texte en deux parties, nous écririons ceci :

Code : Actionscript

```
trace("Hello" + " world !");
```

Il est alors possible de concaténer des chaînes avec des variables de tout type (y compris avec des nombres), et de différentes manières :

Code : Actionscript

```
var coucouDebut:String = "Hello ";
var coucouFin:String = " !";
var monNom:String = "Jérôme";
var monAge:int = 42;
trace(coucouDebut + monNom + coucouFin + " Tu as " + monAge + " ans,
n'est-ce pas ?");
// Affiche : Hello Jérôme ! Tu as 42 ans, n'est-ce pas ?

trace(Number.MIN_VALUE + " à " + Number.MAX_VALUE);
// Affiche : 4.9406564584124654e-324 à 1.79769313486231e+308
```

Quelques variables et fonctions utiles

Ces variables ou fonctions sont obligatoirement attachées à une variable de type `String` à l'aide d'un point. Vous ne pouvez pas les utiliser sur une chaîne de caractères simple, comme `"Hello world !"`.

Longueur d'une chaîne

Pour connaître la longueur d'une chaîne, c'est a dire le nombre de caractères qu'elle contient, il faut utiliser la variable `length` disponible sur notre chaîne, de cette manière :

Code : Actionscript

```
var coucou:String = "Hello world !";
trace("Cette chaîne contient " + coucou.length + " caractères.");
// Affiche : Cette chaîne contient 13 caractères.
```

Le point est important : il signifie que c'est la longueur de cette chaîne particulière que l'on veut. 

Changer la casse d'une chaîne

La casse est l'état d'une lettre, selon si elle est en minuscule ou en majuscule. Il est possible de modifier la casse de l'ensemble d'une chaîne de caractères en utilisant les fonctions `toLowerCase()` (en casse minuscule) et `toUpperCase()` (en casse majuscule) :

Code : Actionscript

```
var coucou:String = "Hello world !";
// En minuscules
trace(coucou.toLowerCase()); // hello world !
// En majuscules
trace(coucou.toUpperCase()); // HELLO WORLD !
```

Rechercher dans une chaîne

Il peut être utile de rechercher un ou plusieurs caractères dans une chaîne. Pour cela, on utilise la fonction `indexOf()` (position de) :

Code : Actionscript

```
maVariable.indexOf("Chaîne recherchée");
```

La fonction renvoie la position du premier caractère de la chaîne recherché dans la variable, ou `-1` si elle ne l'a pas trouvé.

Vous pouvez aussi spécifier une variable de type `String` entre les parenthèses, au lieu d'une chaîne de caractères simple.

Cherchons la position de la première lettre « a » dans notre variable :

Code : Actionscript

```
var coucou:String = "Hello world !";
trace(coucou.indexOf("a")); // -1, on n'a rien trouvé :(
```

Effectivement, il n'y a pas de « a » dans notre chaîne... Retentons notre chance avec le mot « world » :

Code : Actionscript

```
trace(coucou.indexOf("world")); // 6 ! Victoire !
```

Bravo, nous avons trouvé le mot « world » à la 7e lettre !

Il se situe à la 7e position, car **le numéro des caractères commence à zéro** : le premier caractère a le numéro 0, le deuxième a le numéro 1 et ainsi de suite.
Du coup, pour avoir le numéro du dernier caractère dans la chaîne, il faut prendre sa longueur moins un : `coucou.length - 1`.

Remplacer dans une chaîne

Cette fonction `replace()` est similaire à la fonction de recherche, mais il faut en plus indiquer le texte qui va remplacer l'ancien.

Code : Actionscript

```
maVariable.replace("Chaîne recherchée", "Chaîne à insérer à la
place");
```

Modifions dynamiquement notre chaîne pour remplacer « world » par « Jérôme » :

Code : Actionscript

```
trace(coucou.replace("world", "Jérôme")); // Hello Jérôme !
```

La variable de départ n'est pas modifiée : la fonction se contente de *renvoyer* la nouvelle chaîne, que l'on peut afficher ou mettre dans une variable :

Code : Actionscript

```
var coucou:String = "Hello world !";
var salut:String = coucou.replace("world", "Jérôme");
trace(salut); // Hello Jérôme !
trace(coucou); // Hello world !
```

Le texte n'est remplacé qu'une seule fois : dès que le texte recherché est rencontré, il est remplacé et la fonction s'arrête.

Code : Actionscript

```
var coucou:String = "Hello world world world !";
trace(coucou.replace("world", "Jérôme")); // Hello Jérôme world
world !
```

Pour pouvoir remplacer tous les « world » en « Jérôme », il faut faire autant de `replace()` que nécessaire :

Code : Actionscript

```
var coucou:String = "Hello world world world !";
coucou = coucou.replace("world", "Jérôme");
trace(coucou); // Hello Jérôme world world !
coucou = coucou.replace("world", "Jérôme");
trace(coucou); // Hello Jérôme Jérôme world !
coucou = coucou.replace("world", "Jérôme");
trace(coucou); // Hello Jérôme Jérôme Jérôme !
```

Il existe une autre méthode bien plus efficace et puissante qui consiste à utiliser les **expressions régulières** (ou **RegExp**) : ce sont des codes suivant des règles précises, capables de rechercher et de remplacer du texte plusieurs fois dans une même chaîne, parmi une foule d'autres choses fort utiles. Malheureusement, cette notion est plutôt complexe à appréhender, je ne l'aborderais pas directement dans le cours, mais plutôt en annexe.

En résumé

- Une **variable** permet de mémoriser une valeur.
- Le **mot-clé** `var` sert à déclarer une variable, qui peut être affectée d'une valeur par un signe « = ».
- On utilise les trois types `int`, `uint` et `Number` pour des valeurs numériques.
- Pour effectuer des calculs, il existe principalement les **opérateurs** de base : `+`, `-`, `*`, `/` et `%`.
- Grâce à la classe `Math`, il est possible de réaliser des opérations mathématiques complexes.
- Le type `String` est réservé à l'utilisation des chaînes de caractères.
- Différentes fonctions permettent de manipuler les chaînes de caractères, et il est possible de les concaténer grâce à l'opérateur `+`.

Les conditions

Nous avons vu dans le précédent chapitre comment manipuler les nombres à l'aide d'*opérateurs* et de *variables*.
Nous allons voir à présent comment tester le contenu de ces variables ; ainsi vous pourrez exécuter des instructions ou non pour les différentes valeurs que pourront prendre vos variables. Comme vous pourrez le voir, les conditions sont très utiles dans le monde de l'informatique, et sont la base de l'interactivité des machines, sans quoi elles feraient toujours la même chose...

Écriture d'une condition

Qu'est-ce qu'une condition ?

Les conditions permettent de tester le contenu d'une ou plusieurs variables. Ainsi vous pourrez exécuter des instructions différentes suivant le résultat du test. Grâce à ces structures conditionnelles, le programme sera alors en mesure de prendre des décisions. Nos programmes seront donc moins monotones et pourront réagir différemment suivant les circonstances : imaginez si tous les programmes faisaient exactement la même chose quel que soit le contexte, cela serait bien ennuyant !
Voici le genre d'instructions que nous serons capables de réaliser à l'aide des conditions :

Code : Autre

```
SI ma condition est vraie
ALORS effectuer mes instructions
```

Dans une condition, nous pourrons ainsi tester différentes choses. Par exemple, nous pourrions tester un nombre entier pour savoir s'il est positif ou non. Nous cherchons donc une relation entre deux valeurs pour pouvoir les *comparer*. Pour cela nous utiliserons donc divers **opérateurs** présentés dans la suite.
Si la condition est vraie, alors les instructions qui la suivent sont exécutées. Dans le cas contraire, elles sont tout simplement ignorées.

Ne vous inquiétez pas si vous avez du mal à saisir le concept, vous comprendrez mieux au fil du chapitre.

En Actionscript, comme dans beaucoup d'autres langages, les conditions renvoient automatiquement une valeur de type `Boolean` comme nous le verrons plus loin dans le chapitre : **`true`** pour *vraie* et **`false`** pour *fausse*.

Les opérateurs relationnels

Les opérateurs relationnels permettent de comparer une variable à une valeur, ou encore deux variables entre elles. Dans le tableau ci-dessous sont répertoriés les différents symboles correspondant. Ces symboles seront utilisés en permanence, c'est pourquoi il serait judicieux de les retenir.

Opérateur	Signification pour des valeurs numériques	Signification pour des caractères
<	est inférieur à	est avant dans l'ordre alphabétique à
>	est supérieur à	est après dans l'ordre alphabétique à
<=	est inférieur ou égal à	est avant dans l'ordre alphabétique ou identique à
>=	est supérieur ou égal à	est après dans l'ordre alphabétique ou identique à
==	est égal à	est identique à
!=	est différent de	est différent de
===	est strictement égal à	est strictement identique à
!==	est strictement différent de	est strictement différent de

De manière générale, les opérateurs sont utilisés ainsi : une variable ou une valeur, l'opérateur et une autre variable ou valeur.
Voici quelques exemples de conditions :

Code : Actionscript

```
// Cette condition est vraie si monEntier contient un nombre
supérieur à 2
monEntier > 2
// Cette deuxième condition est vraie si maVariable et
monAutreVariable contiennent la même valeur
monVariable == monAutreVariable
```

Vous pouvez également comparer des chaînes de caractères :

Code : Actionscript

```
var t1:String = "Salut";
var t2:String = "Salut";
var t3:String = "Bonjour";
trace(t1 == t2); // Affiche : true
trace(t1 == t3); // Affiche : false
trace(t1 > t3); // Affiche : true, car Salut est après Bonjour dans
l'ordre alphabétique
```

Vous remarquerez que le test d'égalité s'effectue à l'aide de deux signes « = ». Les débutants omettent souvent le deuxième symbole « = », ce qui est source d'erreurs dans votre code. En effet, le symbole « = » seul est un signe d'affectation pour les variables comme nous l'avons vu dans le précédent chapitre.

Comparer deux variables : une question de types

Un opérateur relationnel permet de comparer uniquement deux expressions du même type. Il n'est pas possible de comparer deux variables dont l'une par exemple, serait de type `int` et l'autre de type `String`. Si cela se produisait, le compilateur vous afficherait un message d'erreur de ce type : « Error: Comparison between... ».
Toutefois, il existe une exception : les nombres, qui peuvent être comparés entre eux même si leur type varie entre `int`, `uint` et `Number`.
Par exemple, le code suivant fonctionnera sans erreur :

Code : Actionscript

```
var nombre:Number = 0.4;
var entier:int = 1;
trace(nombre < entier); // Affiche : true
```

Mais alors, comment puis-je facilement comparer un nombre et une chaîne de caractères ?

Et bien, il suffit de transformer l'une de nos deux variables pour que les deux aient exactement le même type ! Par exemple, transformons le nombre en chaîne de caractères :

Code : Actionscript

```
var nombre:Number = 3.14;
var texte:String = "3.14";
trace(nombre.toString() == texte); // Affiche : true
```

En effet, taper `.toString()` derrière notre variable de type `Number` transforme sa valeur en chaîne de caractères afin que la comparaison fonctionne correctement !

La variable `nombre` n'est pas réellement transformée, elle reste de type `Number` par la suite.

Précisions sur les opérateurs stricts

Les opérateurs stricts (=== et !==) servent à comparer deux objets quelconques (par exemple, deux variables), en regardant non seulement leur valeur, mais aussi leur type. Ces opérateurs sont peu utilisés : en général, on connaît à l'avance le type des variables que l'on manipule.
Par exemple, je déclare et initialise trois variables de type `Object`, c'est-à-dire qu'elles n'ont pas de type bien défini et que l'on peut y mettre ce que l'on souhaite ; attention toutefois à ne pas abuser de ce type un peu spécial, il ralentit l'exécution de votre programme. Ensuite, je teste la valeur de ces variables avec ==, pour finalement tester leur valeur et leur type en même temps avec ===. De toute évidence, le caractère « 3 » n'est pas du même type que l'entier 3.

Code : Actionscript

```
var nom:Object = "42";
var prenom:Object = "42";
var age:Object = 42
trace(nom === prenom); // Affiche : true
trace(nom == age); // Affiche : true
trace(nom === age); // Affiche : false
```

Dans ce cas très précis, votre programme Flash convertira automatiquement le format des variables lorsqu'il est nécessaire, sauf si vous utilisez les opérateurs === ou !==. Ainsi la comparaison entre la chaîne de caractères et l'entier se déroule sans accroc.

Les opérateurs logiques

Contrairement aux opérateurs précédents qui permettaient de comparer des valeurs, les opérateurs logiques servent à *combiner plusieurs conditions*. Ceux-ci peuvent avoir leur utilité lorsque nous voulons tester par exemple, si un nombre est compris dans un intervalle. Les opérateurs relationnels ne permettent pas ce genre de comparaison. C'est pourquoi nous pouvons contourner le problème en combinant plusieurs conditions.
Prenons un exemple : nous voulons savoir si une variable `monNombre` est comprise entre 0 et 10. Pour cela nous pourrions décomposer ce test en deux conditions :

- `monNombre` est supérieur à 0
- `monNombre` est inférieur à 10.

Les opérateurs logiques nous permettent alors d'associer ces deux conditions en une seule : `monNombre > 0 ET monNombre < 10`.

Le tableau ci-dessous présente donc ces différents opérateurs, ainsi que les symboles qui leur sont associés :

Opérateur	Signification
!	NON logique
&&	ET logique
||	OU logique

Ainsi, nous allons comparer notre nombre par rapport à un intervalle à l'aide de l'opérateur `&&` :

Code : Actionscript

```
// Test pour savoir si monNombre est compris entre 0 et 10
monNombre > 0 && monNombre < 10
```

La priorité des opérateurs

Pour décider dans quel ordre les différentes opérations seront effectuées, les opérateurs respectent les règles de priorité suivantes :

- les opérateurs arithmétiques (+, -, *, /, %) sont prioritaires par rapport aux opérateurs relationnels (==, !=, <, >, etc...) ;
- les opérateurs relationnels sont prioritaires par rapport aux opérateurs logiques (!, && et ||) ;
- les opérations entourées de parenthèses sont toujours prioritaires.

Ainsi la condition `3 + 4 > 5` est vraie du fait que l'addition est effectuée avant la comparaison.

Quelques exemples d'opérations imbriquées :

Code : Actionscript

```
trace(1 == 2); // Affiche : false
trace(1 + 1 == 2); // Affiche : true
trace(1 == 2 || 2 == 2); // Affiche : true
trace(1 != 2 && 1 + 1 == 2); // Affiche : true
trace(1 != 2 && 1 + 2 == 2); // Affiche : false
```

Pour mieux comprendre l'utilisation de ces conditions, nous allons étudier différentes structures qui les utilisent. Commençons tout de suite avec l'instruction **`if...else`**.

L'instruction if...else

La structure de base

La syntaxe

L'instruction **`if...else`** est la structure de base des conditions. Grâce à elle, nous pouvons exécuter des instructions différentes suivant si la condition est vraie ou fausse. Sans plus attendre, voyons la structure **`if...else`** écrite en Actionscript :

Code : Actionscript

```
if(/* Condition */)
{
    // Instructions si la condition est vraie
}
else
{
    // Instructions si la condition est fausse
}
```

Vous remarquerez donc dans cette structure, les deux blocs d'instructions définis par les paires d'accolades. Si la condition est vérifiée alors le premier bloc d'instructions sera exécuté, *sinon* (traduction du mot « else ») ce sera le second.
Bien entendu, l'écriture du bloc **`else`** n'est pas obligatoire. En revanche, il n'est pas possible d'écrire un bloc **`else`** seul. Vous pourriez donc vous contenter du code suivant :

Code : Actionscript

```
if(/* Condition */)
{
    // Instructions si la condition est vraie
}
```

Parce que rien ne vaut un bon exemple, nous allons tout de suite tester une première condition dans notre IDE. Pour cela, nous prendrons une variable nommée `maVie` représentant par exemple, la vie restante dans un jeu quelconque. Nous pourrions alors tester si vous êtes encore en vie ou si vous êtes mort. Voici donc le code à insérer juste après le commentaire `// entry point` de votre classe `Main` :

Code : Actionscript

```
var maVie:uint = 1;

if(maVie == 0)
{
    trace("Vous êtes mort.");
}
else
{
    trace("Vous êtes toujours en vie.");
}

// Affiche : Vous êtes toujours en vie.
```

Ici la variable `maVie` est égale à 1, et donc la condition `maVie == 0` est fausse. Ainsi lors de l'exécution, seule l'instruction à l'intérieur des accolades du **`else`** sera exécutée. Vous verrez donc apparaitre le message : « Vous êtes toujours en vie. ».
N'hésitez pas à tester ce code avec différentes valeurs pour `maVie` afin de bien comprendre le fonctionnement.

Pour écrire une condition **`if`** seule ou **`if...else`**, il faut respecter un certain nombre de règles et de convention que nous nous apprêtons à détailler.

Les règles et conventions d'écriture

Nous allons maintenant définir quelles sont les différentes règles d'écriture ainsi que des conventions fortement recommandées.
Tout d'abord, vous aurez sûrement remarqué l'absence de point-virgule « ; » après les accolades. En effet, les conditions ne se terminent jamais par un point-virgule. D'autre part, les parenthèses qui entourent votre condition sont *obligatoires*.
En revanche, les accolades peuvent être facultatives mais *uniquement* dans un cas ; il s'agit du cas où il n'y a qu'une seule instruction à l'intérieur du bloc d'instructions. Il est alors possible d'écrire votre condition suivant l'une des trois formes ci-dessous :

Code : Actionscript

```
if(maVie == 0)
    trace("Vous êtes mort.");
else
    trace("Vous êtes toujours en vie.");
```

ou bien :

Code : Actionscript

```
if(maVie == 0)
    trace("Vous êtes mort.");
else
{
    trace("Vous êtes toujours en vie.");
}
```

ou encore :

Code : Actionscript

```
if(maVie == 0)
```

```
{
    trace("Vous êtes mort.");
}
else
    trace("Vous êtes toujours en vie.");
```

Pour finir ce paragraphe, nous parlerons d'une chose *très importante* en termes de présentation. Vous aurez certainement remarqué la mise en forme utilisée depuis le début pour écrire les conditions :

- l'accolade ouvrante sur une ligne
- vos instructions décalées vers la droite à l'aide d'une tabulation
- l'accolade fermante sur une ligne

Cette mise en forme est appelée l'*indentation* et n'est pas obligatoire, cependant il est *très fortement déconseillé* de tout écrire sur une seule ligne ou de ne pas décaler les instructions à l'intérieur des accolades. En effet, respecter ces règles vous permettra de rendre vos codes beaucoup plus *clairs* et *lisibles* par vous mais aussi par d'autres personnes qui ne connaîtront pas la façon dont vous avez réalisé votre programme.

Les ternaires

Dans certains cas il est possible de condenser des conditions : On appelle cela les **expressions ternaires**.
Cette technique n'est pas applicable tout le temps, et il ne faut pas en abuser. Pour introduire cette notion, nous allons partir du code suivant :

Code : Actionscript

```
var monMessage:String;
if(maVie == 0)
    monMessage = "Vous êtes mort.";
else
    monMessage = "Vous êtes toujours en vie.";
trace(monMessage);
```

Vous remarquerez qu'ici la condition sert *uniquement* à affecter une variable d'une valeur qui dépend d'une condition. Les expressions ternaires ne peuvent être utilisées que dans ce cas-là !
Il est alors possible de réécrire le code précédent sous une forme condensée :

Code : Actionscript

```
var monMessage:String = (maVie == 0) ? "Vous êtes mort." : "Vous
êtes toujours en vie.";
trace(monMessage);
```

Le principe est alors d'écrire l'ensemble de la condition en *une seule ligne* et d'affecter directement la variable. C'est une forme d'écriture qui peut être pratique, mais sachez qu'en réalité elle est très peu utilisée car elle est difficile à lire.
Utilisez donc les expressions ternaires pour diminuer le volume de votre code, mais tâchez de garder un code qui soit le plus facilement lisible !

L'écriture des expressions ternaires est assez spéciale et souvent trompeuse. Notez donc la présence du symbole « ? » qui permet de tester la condition placée juste avant, ainsi que le signe « : » permettant de remplacer le mot-clé **else** et de séparer les instructions.

Le type booléen

En Actionscript, il existe un type de variable de type booléen : `Boolean`. Ces variables ne peuvent prendre que deux valeurs : **true** ou **false**. Les booléens sont très utilisés avec les conditions car ils permettent facilement d'exprimer si une condition est vraie ou fausse. D'ailleurs une condition est elle-même exprimée sous la forme d'un booléen. Pour s'en assurer, vous allez tester le code suivant :

Code : Actionscript

```
if(true)
    trace("Ma condition est vraie.");
else
    trace("Ma condition est fausse.");

// Affiche : Ma condition est vraie.
```

Nous pouvons remarquer qu'en effet la condition **true** est vraie, c'est pourquoi l'utilisation de variables de type booléen peut être préférable. Il n'est alors plus nécessaire d'écrire le test « == **true** » pour une variable de type booléen.
Ainsi, si vous utilisez des noms judicieux pour vos variables, le code paraîtra plus clair :

Code : Actionscript

```
var estEnVie:Boolean = true;

if(estEnVie)
{
    trace("Vous êtes toujours en vie.");
}
else
{
    trace("Vous êtes mort.");
}

// Affiche : Vous êtes toujours en vie.
```

Étant donné que le résultat d'une condition est un booléen, il est tout à fait possible de le stocker dans une variable :

Code : Actionscript

```
var maVie:uint = 1;
var estEnVie:Boolean = (maVie > 0);
trace(estEnVie); // Affiche : true
```

L'utilisation des booléens sert principalement à alléger votre code et le rendre plus lisible. Encore une fois, choisissez des noms explicites pour vos variables, cela permettra de faire plus facilement ressortir la logique de votre programme.

La structure avec `else if`

Avant de clore cette partie sur la structure **if...else**, nous allons voir comment réaliser des tests supplémentaires avec **else if**. Ici rien de bien compliqué, c'est pourquoi vous comprendrez aisément le code ci-dessous :

Code : Actionscript

```
var monNombre:int = 1;

if(monNombre < 0)
{
    trace("Ce nombre est négatif.");
}
else if(monNombre == 0)
{
    trace("Ce nombre est nul.");
}
else
{
    trace("Ce nombre est positif.");
```

```
}

// Affiche : Ce nombre est positif.
```

Bien qu'il soit possible d'effectuer plusieurs tests à la suite à l'aide de cette méthode, cela peut s'avérer assez répétitif. C'est pourquoi, il existe une structure qui permet d'alléger l'écriture de telles conditions : il s'agit de l'instruction **switch** que nous allons détailler dans la suite.

L'instruction switch

L'utilisation conventionnelle

Face à la structure **if...else**, la condition **switch** permet de simplifier et d'alléger le code lorsque vous voulez tester différentes valeurs pour une même variable. Cette structure n'offre pas plus de possibilités que celle en **if...else**, il s'agit simplement d'une manière différente d'écrire des conditions.
Voici comment se présente l'instruction **switch** :

Code : Actionscript

```
switch(/* Variable */)
{
    case /* Argument */ :
        // Instructions
        break;
    case /* Argument */ :
        // Instructions
        break;
    case /* Argument */ :
        // Instructions
        break;
    default :
        // Instructions
}
```

Pour utiliser cette structure, il suffit de renseigner la variable à tester puis d'étudier les différents arguments. Notez la présence de l'instruction « **break;** » qui permet de sortir du **switch**. Cette instruction est *obligatoire*, en cas d'absence les instructions situées en dessous seraient exécutées. Enfin, le **default** correspond au cas par défaut, nous pouvons le comparer au **else** d'une instruction **if...else**.
Voici un exemple de code que vous pouvez tester :

Code : Actionscript

```
var monNombre:int = 1;
switch(monNombre)
{
    case 0 :
        trace("Ce nombre vaut 0.");
        break;
    case 1 :
        trace("Ce nombre vaut 1.");
        break;
    case 2 :
        trace("Ce nombre vaut 2.");
        break;
    default :
        trace("Ce nombre ne vaut ni 0, ni 1, ni 2.");
}
```

Notez qu'après chaque argument vous devez écrire un double point et non un point-virgule.

Une utilisation spécifique à l'Actionscript

Contrairement à d'autres langages tels que le C ou encore le Java, l'instruction **switch** en Actionscript ne permet pas uniquement de faire des égalités. En plaçant la valeur **true** à la place de la variable, il est alors possible de la comparer avec un autre booléen, en particulier une condition.
Voici la structure **switch** pouvant tester diverses conditions :

Code : Actionscript

```
switch(true)
{
    case (/* Condition */) :
        // Instructions
        break;
    case (/* Condition */) :
        // Instructions
        break;
    case (/* Condition */) :
        // Instructions
        break;
}
```

Voici un exemple concret qui permet de tester si une variable `monNombre` est un nombre négatif, nul ou positif :

Code : Actionscript

```
var monNombre:int = 1;
switch(true)
{
    case (monNombre < 0) :
        trace("Ce nombre est négatif.");
        break;
    case (monNombre == 0) :
        trace("Ce nombre est nul.");
        break;
    case (monNombre > 0) :
        trace("Ce nombre est positif.");
        break;
}
```

En résumé

- Grâce aux **conditions**, nous pouvons effectuer des choses différentes suivant l'état d'une ou plusieurs variables.
- Les conditions sont réalisées à l'aide d'**opérateurs relationnels** et **logiques**, dont le résultat est un **booléen**.
- L'instruction **if...else** est la plus utilisée.
- Les **expressions ternaires** permettent de condenser l'écriture d'une instruction **if...else**.
- Lorsque vous avez une multitude de valeurs à tester, l'instruction **switch** est plus appropriée.

Les boucles

Dans le chapitre précédent, nous avons vu comment réaliser des **conditions** et permettre ainsi aux machines de faire des choix. C'est déjà un grand pas dans le monde informatique mais ce n'est pas suffisant : à présent, nous allons introduire les **boucles** !

Les boucles sont liées à la notion de **répétitivité** : il s'agit en fait de réitérer une série d'instructions plusieurs fois. Les boucles sont indispensables dans tout un tas de situations, par exemple lorsque nous demandons à l'utilisateur de jouer tant que la partie n'est pas terminée. Pour cela nous disposons de trois boucles en Actionscript 3 que nous détaillerons au fil du chapitre.
Étant donné que nous avons déjà vu comment écrire une condition, ce chapitre devrait être relativement simple à appréhender.

La boucle while

Le principe

Comme nous l'avons dit précédemment, les boucles permettent de répéter des instructions un certain nombre de fois. La difficulté est alors de savoir quand ces instructions doivent cesser d'être répétées. Pour cela, nous aurons besoin des *conditions*.
Sans plus attendre, voici le genre d'instructions faisables à l'aide des boucles :

Code : Autre

```
TANT QUE ma condition est vraie
ALORS effectuer mes instructions
```

Dans une boucle **`while`**, la condition s'utilise exactement de la même manière que pour une instruction **`if...else`**. Il faudra donc déclarer une variable avant la boucle puis la tester à l'intérieur de la condition. En revanche, dans ce cas la variable devra être *mise à jour à l'intérieur* de la boucle pour pouvoir en sortir. En effet, pour stopper la répétition des instructions, la condition doit obligatoirement *devenir fausse* à un certain moment.
Oublier de mettre à jour cette variable conduirait à une boucle infinie, c'est-à-dire que les instructions se répèteraient à l'infini.

Dans le cas d'une boucle **`while`**, la condition est testée avant d'entrer dans la boucle. Ainsi si la condition est déjà *fausse* avant la boucle, les instructions à l'intérieur de celle-ci ne seront *jamais exécutées*.

Encore une fois, vous comprendrez mieux le fonctionnement des boucles au fur et à mesure de la lecture de ce chapitre.

Écriture en Actionscript

En Actionscript, la boucle **`while`** permet de gérer toutes les situations : celles que nous verrons plus tard sont légèrement différentes mais n'offrent aucune fonctionnalité supplémentaire. La structure d'une boucle est très similaire à celle des structures conditionnelles vues précédemment.
Voici donc comment écrire une boucle **`while`** en Actionscript :

Code : Actionscript

```
while(/* Condition */)
{
    // Instructions si vraie
}
```

Comme pour l'instruction **`if...else`**, les accolades deviennent facultatives s'il n'y a qu'une seule condition :

Code : Actionscript

```
while(/* Condition */)
    // Instruction si vraie
```

Cette seconde syntaxe est correcte : en écrivant cela vous n'aurez pas de message d'erreur. Cependant dans une boucle digne d'intérêt, vous n'aurez en réalité que très rarement une seule instruction. En effet, votre boucle contiendra toujours au moins une instruction ainsi qu'une mise à jour de votre variable.

Afin de mieux comprendre le fonctionnement et la syntaxe d'une boucle **while**, nous allons voir différents exemples dans la suite.

Exemples d'utilisation

Pour commencer, nous allons tester une boucle qui affiche la valeur de notre variable testée dans la condition. Cet exemple permettra de mieux comprendre le fonctionnement d'une boucle.
Voici le code source de notre boucle à tester :

Code : Actionscript

```
var i:uint = 0;
while(i < 5)
{
    trace(i);
    i = i + 1;
}
```

Si vous lancez le projet, vous devriez obtenir ceci dans la console de sortie :

Code : Console

```
0
1
2
3
4
```

Vous constaterez alors que la valeur `5` n'est pas affichée. Effectivement, lorsque notre variable `i` prend cette valeur, la condition `i < 5` n'est plus vérifiée ; l'instruction `trace(i)` n'est donc plus exécutée. Il faudra ainsi être vigilant sur l'écriture de votre condition pour que votre boucle s'exécute bien comme vous le désirez.

L'intérêt d'utiliser une boucle **while** est que nous n'avons pas besoin de connaître à l'avance ou de nous soucier du nombre de répétitions. Pour illustrer cela nous allons prendre un exemple : imaginons que nous cherchions la puissance de 2 correspondant à 1024. Nous devons alors calculer les puissances successives de 2 jusqu'à atteindre 1024. Cependant ici nous ne savons pas combien de fois nous allons devoir répéter l'opération. Nous utiliserons donc une boucle **while** de la manière suivante :

Code : Actionscript

```
var monNombre:uint = 2;
var i:uint = 1;
while(monNombre != 1024)
{
    i = i + 1;
    monNombre *= 2;
}
trace(i); // Affiche : 10
```

En effet, pour ceux qui connaissaient déjà la réponse : $2^{10} = 1024$. Notre boucle s'est donc exécutée 9 fois ce qui correspond à `i = 10`. Nous voyons bien ici que nous n'avions pas besoin de connaître à l'avance le nombre de répétitions de la boucle.

Quelques astuces

Vous savez déjà tout ce qu'il faut savoir sur la boucle **while**, cependant nous allons voir ici quelques astuces pour améliorer la lisibilité de votre code. Ces quelques conseils ne sont en aucun cas une obligation, il s'agit principalement de pratiques très courantes au sein des programmeurs. Ceci permettra d'alléger votre code source mais également de le rendre plus lisible pour un autre programmeur.
Voici donc quelques astuces fortement recommandées :

- traditionnellement en Actionscript et dans beaucoup d'autres langages, on utilise les noms de variables `i`, `j` et `k` pour compter à l'intérieur d'une boucle. En utilisant ces noms, vous facilitez la lecture de votre code, surtout si celui-ci est assez conséquent.
- lorsque vous utiliserez des boucles, vous serez très souvent amené à incrémenter ou décrémenter vos variables : `i = i + 1` ou `i = i- 1`. Pour simplifier l'écriture, nous pouvons utiliser pour l'incrémentation `i++` et pour la décrémentation `i--`.

Enfin pour finir, nous allons revenir sur la notion de boucle infinie. Comme nous l'avons dit, il s'agit d'une boucle dont la condition est toujours vérifiée. Les instructions à l'intérieur sont alors répétées à l'infini.
En voici un exemple :

Code : Actionscript

```
var maCondition:Boolean = true;
while(maCondition)
{
    trace("Ceci est une boucle infinie");
}
```

Vous vous en doutez certainement, cette boucle va afficher le message : « Ceci est une boucle infinie » un nombre indéfini de fois, comme ceci :

Code : Console

```
Ceci est une boucle infinie
Ceci est une boucle infinie
Ceci est une boucle infinie
...
```

En général, il faut à tout prix éviter d'écrire ce genre de choses. Cependant ce type de boucles peut s'avérer utile dans certains cas.

La boucle do...while

Nous allons à présent parler de la boucle **`do...while`**, qui est une voisine très proche de la boucle **`while`**. Toutefois nous pouvons noter une différence dans la syntaxe par l'apparition du mot **`do`** ainsi que du transfert de la *condition* en *fin de boucle*. Ceci apporte une légère différence dans l'interprétation du code mais nous y reviendrons après.
Voici la syntaxe de cette nouvelle boucle :

Code : Actionscript

```
do
{
    // Instructions si vraie
} while(/* Condition */);
```

Notez cependant l'apparition d'un point-virgule « ; » après la condition. L'oubli de ce point-virgule est une erreur très courante chez les débutants. Veillez donc à ne pas commettre cette erreur, ce qui vous évitera des problèmes de compilation.

Quelle est la différence avec la boucle précédente ?

Dans une boucle **`while`** classique, la condition est en début de boucle. C'est pourquoi la boucle peut ne jamais être exécutée si la condition est fausse dès le départ. En revanche dans une boucle **`do...while`**, la condition est placée à la fin de la boucle. De ce fait, la boucle s'exécutera *toujours au moins une fois*. Il est donc possible d'utiliser cette boucle pour s'assurer que le programme rentrera au moins une fois dans la boucle.
Voici un exemple de calcul de PGCD :

Code : Actionscript

```
var nombre1:uint = 556;
var nombre2:uint = 148;
do
{
    var reste:uint = nombre1 % nombre2;
    nombre1 = nombre2;
    nombre2 = reste;
} while(nombre2 != 0);
trace("PGCD = " + nombre1); // Affiche : PGCD = 4
```

Petit rappel : PGCD pour *Plus Grand Commun Diviseur*. Si vous ne vous souvenez plus de l'algorithme, il faut diviser le plus grand nombre par le plus petit. Ensuite on prend le reste de la division euclidienne pour diviser le nombre le plus petit. Puis on reprend le dernier reste pour diviser le reste de la division précédente. Et ainsi de suite jusqu'à ce qu'un reste soit nul. Vous avez ainsi votre PGCD qui correspond au dernier reste non nul !

Encore ici, vous pouvez remarquer qu'il n'est pas nécessaire de connaître le nombre d'exécutions de la boucle contrairement à une boucle **for** dont nous allons parler maintenant.

La boucle for

Présentation

La syntaxe

Comme nous l'avons déjà dit, la boucle **while** permet de réaliser toute sorte de boucle. Cependant son écriture est quelque peu fastidieuse, c'est pourquoi il existe une écriture condensée utile dans certains cas : la boucle **for**. Celle-ci est utilisée lorsque nous connaissons à l'avance le nombre d'exécutions de la boucle, soit directement soit à travers une variable. Cette boucle permet de concentrer le traitement de la variable utilisée, rappelez-vous avec une boucle **while**, nous devions passer par les étapes suivantes :

- déclaration et initialisation de la variable avant la boucle
- condition sur la variable en entrée de boucle
- incrémentation de la variable à l'intérieur de la boucle.

Grâce à la boucle **for**, nous pouvons maintenant condenser ces trois étapes au même endroit :

Code : Actionscript

```
for(/* Initialisation */; /* Condition */; /* Incrémentation */)
{
    // Instructions
}
```

Attention là encore à la présence des points-virgules « ; » pour séparer chaque champ. C'est également une source de problèmes pour la compilation.

Quelques exemples

La boucle **for** est très utilisée en programmation, elle permet d'éviter d'avoir à écrire des instructions répétitives mais permet également de faire des recherches dans les tableaux que nous aborderons très bientôt. En attendant, voici quelques exemples d'utilisation de la boucle **for** :

Code : Actionscript

```
for(var i:int = 0; i < 10; i++)
{
    trace("La valeur de i est " + i);
}
```

Ici également, les accolades deviennent facultatives s'il n'y a qu'une seule instruction. Aussi il est possible d'utiliser la boucle **for** en décrémentation ou en encore par pas de 10 si vous le désirez :

Code : Actionscript

```
for(var i:int = 10; i > 0; i--)
    trace("La valeur de i est " + i);
```

La boucle **for** n'a rien de difficile, il s'agit simplement d'une boucle **while** condensée. En revanche la syntaxe est assez spéciale, c'est pourquoi il est nécessaire de la retenir car cette boucle vous servira plus que vous ne l'imaginez ! 

En résumé

- Les **boucles** permettent de répéter une série d'instructions tant qu'une condition est vérifiée.
- La boucle **while** permet de tout faire mais il est possible que les instructions à l'intérieur ne soient *jamais* exécutées.
- La boucle **do...while** est identique à la précédente, néanmoins la boucle est exécutée *au moins une fois*.
- La boucle **for** est une *forme condensée* du **while** dans le cas où on *connaît* le nombre de répétitions de la boucle.
- L'incrémentation est utilisée en permanence dans les boucles et son écriture peut être simplifiée par les expressions « `i++` » et « `i--` ».

Les fonctions

Maintenant nous allons introduire les **fonctions** !
Sachez que c'est un concept qui existe dans tous les langages de programmation et que ces fonctions servent à *structurer* notre programme en petits bouts.

Nous verrons en quoi consiste réellement ces fonctions dont nous avons déjà entendu parler. Nous apprendrons comment utiliser des fonctions toutes prêtes, mais aussi comment en créer nous-mêmes. Pour bien comprendre, nous présenterons quelques exemples qui utilisent également les connaissances acquises jusqu'ici.

Concept de fonction

Le principe de fonctionnement

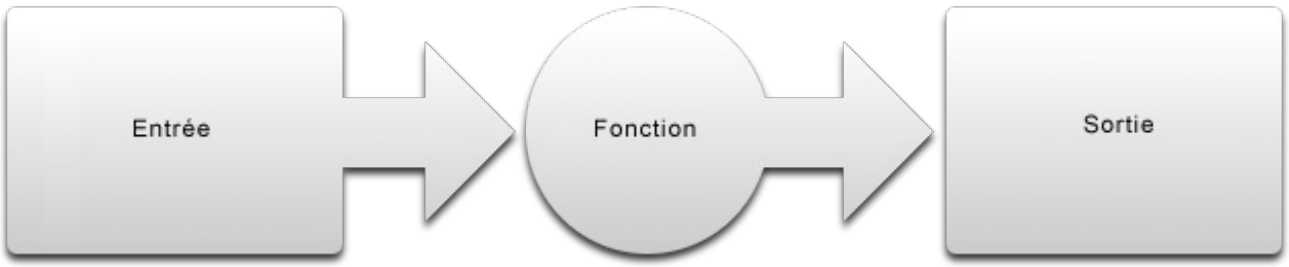
Depuis le début de ce cours, tout ce que nous faisons se trouve à l'intérieur d'une fonction nommée `Main` (ou peut-être d'une fonction `init` si vous utilisez *FlashDevelop*). En effet, nous n'avons pas encore écrit à l'extérieur des accolades de celle-ci. En général pour écrire un programme, il faut compter entre une centaine de lignes de code à plusieurs milliers pour les plus conséquents. Il n'est donc pas envisageable d'insérer la totalité du code à l'intérieur de la fonction `Main`. En effet il faut *organiser* le code, ce qui permet de le rendre *plus lisible* mais également de *faciliter le débogage*. Pour cela nous allons découper notre programme en morceaux. Nous regrouperons donc certaines instructions ensemble pour créer des bouts de programme, puis nous les assemblerons pour obtenir le programme complet. Chaque morceau est alors appelé : **fonction** !

Comment se présente une fonction ?

Une fonction est une série d'instructions qui ont été regroupées pour une tâche commune. Nous créons alors une fonction que nous appellerons à chaque fois que nous en avons besoin. Celle-ci se compose de la manière suivante :

- une ou des entrées : il s'agit d'informations qui seront utiles au cours de l'exécution de la fonction
- le traitement et les calculs : c'est le cœur de la fonction, les instructions internes sont exécutées
- une *unique* sortie : une fois les instructions de la fonction terminées, celle-ci renvoie un résultat.

Pour résumer tout ceci, voici un schéma reprenant le principe de fonctionnement :



Principe d'une fonction

Notez qu'une fonction ne possède pas *obligatoirement* une entrée *et* une sortie. Celle-ci peut très bien se satisfaire uniquement d'une entrée ou d'une sortie, et plus exceptionnellement aucune des deux. Nous reviendrons là-dessus dans très peu de temps.

En règle générale, on associe à une fonction une tâche bien précise. Par exemple, nous pourrions imaginer une fonction qui calcule et renvoie le carré d'un nombre placé en entrée. Voici un schéma de ce que cela donnerait :

Exemple d'une fonction

Bien évidemment dans un programme, les fonctions sont généralement plus complexes. L'Actionscript possède déjà des centaines de fonctions qui permettent par exemple d'afficher une image à l'écran ou encore de la transformer. Nous apprendrons à les utiliser ainsi qu'à créer nos propres fonctions.

Présentation

Vous ne vous en doutez peut-être pas, mais vous avez déjà utilisé des fonctions depuis le début de ce cours !
Effectivement, nous avons déjà utilisé plusieurs fois la fonction `trace()` et également introduit la fonction `toString()`.
Néanmoins ces deux fonctions sont légèrement différentes dans leur utilisation. Nous présenterons donc brièvement ici ces différences entre les *fonctions dites de base* telle que la fonction `trace()` et les *méthodes* comme la fonction `toString()`.
Allons-y pour les fonctions de base !

Les fonctions de base

L'appel d'une fonction telle que la fonction `trace()` se fait de manière la plus basique qu'il soit. Rappelez-vous la façon dont nous écrivions l'instruction faisant appel à cette fonction :

Code : Actionscript

```
trace("Texte à afficher");
```

Vous remarquerez qu'il suffit simplement d'écrire le nom de la fonction, ainsi que le texte à afficher entre parenthèses.
En revanche ceci est différent pour les méthodes !

Les méthodes

Les méthodes sont liées à la notion de *POO* ou *Programmation Orientée Objet*, je vous rappelle que nous y reviendrons dans une partie entièrement consacrée à ce concept.
Contrairement aux fonctions de base, les méthodes sont associées à un objet. Pour vous montrer ceci, reprenons l'exemple d'utilisation de la fonction `toString()` :

Code : Actionscript

```
var nombre:Number = 3.14;
trace(nombre.toString() == "3.14");
```

Ce qu'il faut noter par rapport à la fonction `trace()`, c'est que la méthode `toString()` est liée à l'objet `nombre` qui est de type `Number`. Notez le point « `.` » séparant l'objet `nombre` de la méthode `toString()` ; il signifie que la méthode est liée à l'objet et qu'elle *ne peut pas* être utilisée indépendamment de l'objet auquel elle est associée.
Quoi qu'il en soit, vous aurez tout le temps de vous familiariser avec ceci au cours de la partie sur la Programmation Orientée Objet.

Création et appel de fonctions

Nous allons maintenant apprendre à créer et appeler des fonctions !
Nous verrons qu'il existe deux façons de déclarer une fonction : les *instructions de fonction* et les *expressions de fonction*. Sans plus tarder, nous allons découvrir tout cela en commençant par les instructions de fonction.

Tout ce que nous allons voir ici concerne les fonctions de base. Néanmoins, les méthodes ne sont pas très différentes et reprendront tous les principes vus tout au long du chapitre. Aussi, soyez attentifs à tout ce qui sera dit ici !
D'autre part, pour l'instant nous nous contenterons de tout écrire à l'intérieur de la fonction `Main` tel que nous le faisons depuis le début de ce tutoriel.

Instructions de fonction

Déclaration

L'*instruction de fonction* est la première technique permettant de définir une fonction. Il s'agit sans aucun doute de la manière que vous serez le plus amenés à utiliser à l'intérieur de vos programmes. Celle-ci débute par le mot-clé **`function`** suivi du nom donné à cette fonction, d'ailleurs voici sa structure :

Code : Actionscript

```
function nomDeLaFonction(sesParametres):sonType
{
    // Instructions à exécuter lors de l'appel de la fonction
}
```

Pour faciliter la compréhension de tous ces champs, nous allons reprendre le modèle utilisé plus haut pour décrire une fonction :

- une ou des entrées : il s'agit des paramètres placés en entrées, par ex: `(monParam1:int, monParam2:String)`. Vous pouvez également ne renseigner aucun paramètre, vous laisserez alors les parenthèses vides, comme ceci `()`.
- le traitement et les calculs : cela correspond aux instructions placées entre les accolades de la fonction, qui seront exécutées à l'appel de la fonction.
- une unique sortie : il est possible que la fonction ne renvoie aucune valeur, auquel cas utilisez le mot-clé `void` comme type de renvoi. Dans le cas contraire, celle-ci ne peut renvoyer qu'une *unique* variable. Le type de cette variable doit être renseigné après les paramètres et précédé de deux-points « `:` ». Le renvoi de cette valeur s'effectue en utilisant le mot-clé **`return`** suivi de la valeur ou variable à renvoyer.

Étant donné que rien ne remplace un bon exemple, voici notre fonction qui élève un nombre au carré :

Code : Actionscript

```
function carre(nombre:int):int
{
    var resultat:int = 0;
    resultat = nombre*nombre;
    return resultat;
}
```

Une variable déclarée à l'intérieur d'une fonction n'est définie que pour cette fonction. À la fin de celle-ci, la variable est supprimée de la mémoire, et n'est donc plus accessible !
Ne tentez donc pas de récupérer sa valeur en dehors des accolades de la fonction !

Sachez qu'il est possible d'effectuer des opérations après le mot-clé **`return`**, il est alors préférable de placer des parenthèses autour du calcul. Veillez cependant à ne pas en faire trop, le code doit toujours rester aéré et lisible.
La fonction précédente aurait donc pu être écrite plus succinctement, de la façon suivante :

Code : Actionscript

```
function carre(nombre:int):int
{
    return (nombre*nombre);
}
```

Voilà notre fonction est définie, il ne reste plus qu'à l'appeler !

Appel

En ce qui concerne l'appel de la fonction, il n'y a rien de bien compliqué sachant que vous avez déjà réalisé cette opération !
De la même manière que pour la fonction `trace()`, nous allons maintenant appeler notre fonction `carre()` en renseignant en paramètre le nombre que nous désirons élever au carré. La fonction nous renvoie une variable de type `int`, nous allons ainsi la stocker dans une variable `resultat` déclarée et de type `int` également.
Voici donc comment procéder :

Code : Actionscript

```
var resultat:int = carre(4);
trace("le carré de 4 est " + resultat); // Affiche : le carré de 4
est 16
```

Lorsque votre fonction ne renvoie rien, le mot-clé `void` peut être facultatif. Cependant je vous conseille *fortement* de l'écrire quand même. D'ailleurs le compilateur vous recommandera également de préciser son type de renvoie : `Warning: return value for function 'nomDeLaFonction' has no type declaration.`

Ce n'est pas très flagrant dans ce cas, mais l'utilisation des fonctions permet d'*organiser* le code et de le rentre *plus lisible*. Ceci sera effectivement le cas lorsque vous aurez des fonctions complexes, et que vous n'aurez plus qu'à les utiliser en les appelant en une seule instruction !
Nous verrons diverses fonctions pour mieux comprendre juste après les expressions de fonction.

Expressions de fonction

Les expressions de fonction sont beaucoup moins utilisées que les instructions de fonction et sont légèrement plus complexes. Les expressions de fonction utilisent une instruction d'affectation pour écrire une fonction. Pour cela, il nous faut définir une fonction anonyme qui sera stockée dans une variable de type `Function`.
Pour éviter de vous noyer dans des explications, voici la structure d'une déclaration d'expression de fonction :

Code : Actionscript

```
var nomDeLaFonction:Function = function (sesParametres)
{
     // Instructions à exécuter lors de l'appel de la fonction
};
```

Nous ne nous attarderons pas trop sur ces expressions de fonction, voici tout de même un exemple d'utilisation :

Code : Actionscript

```
var afficher:Function = function (texte:String):void
{
     trace(texte);
}
afficher("Ceci est la fonction afficher.");
```

Pour l'instant vous pouvez vous contenter uniquement des *instructions de fonction*, ne vous prenez donc pas la tête avec les expressions de fonction qui sont moins utilisées. En revanche retenez bien comment utiliser ces instructions de fonction que nous utiliserons désormais tout le temps.

Quelques exemples

Enfin pour clore ce chapitre sur les fonctions, nous verrons quelques exemples de fonctions qui pourront pour certaines vous être utile à l'avenir.

Message de bienvenue

Voici une fonction qui se contente d'afficher un message de bienvenue à la personne indiquée en paramètre. Cette fonction ne renvoie aucune valeur.
Voici la fonction en question :

Code : Actionscript

```
function bienvenue(nom:String):void
{
    trace("Bonjour " + nom + " et bienvenue sur le Site du Zéro !");
}
bienvenue("Marcel Dupont"); // Affiche : Bonjour Marcel Dupont et
bienvenue sur le Site du Zéro !
```

Pour que votre code soit encore plus lisible, n'hésitez pas à donner des noms explicites à vos fonctions !

Calcul de PGCD

Dans cet exemple, nous allons calculer le PGCD entre deux nombres, comme nous avons appris à le calculer dans le chapitre précédent :

Code : Actionscript

```
function calculPGCD(nombre1:int, nombre2:int):int
{
    do
    {
        var reste:uint = nombre1 % nombre2;
        nombre1 = nombre2;
        nombre2 = reste;
    } while(nombre2 != 0);
    return nombre1;
}
trace("Le PGCD de 56 et 42 est : " + calculPGCD(56,42)); // Affiche
: Le PGCD de 56 et 42 est : 14
```

Pour ceux qui voudraient réellement utiliser cette fonction, sachez qu'ici le `nombre1` est supposé plus grand que le `nombre2`. Si vous le désirez, n'hésitez pas à rajouter une condition pour tester si c'est effectivement le cas !

Calcul d'un maximum

Voici une fonction qui pourrait vous être utile, celle-ci vous renvoie le plus grand nombre entre ces deux paramètres. Il n'y a rien de très compliqué, voici comment procéder :

Code : Actionscript

```
function max(nombre1:int, nombre2:int):int
{
    var resultat:int = 0;
    if(nombre1 > nombre2)
    {
        resultat = nombre1;
    }
    else
    {
        resultat = nombre2;
    }
    return resultat;
}
var nombre1:int = 2;
var nombre2:int = 3;
trace("Le maximum entre " + nombre1 + " et " + nombre2 + " est " +
max(nombre1,nombre2)); // Affiche : Le maximum entre 2 et 3 est 3
```

Lorsque vous réaliserez des programmes assez conséquents, vous devrez économiser au maximum les instructions dans vos programmes pour gagner en performance ou simplement pour le rendre plus lisible. Il peut alors devenir intéressant de regarder si vos fonctions ne peuvent pas s'écrire différemment mais surtout en moins de lignes. Ceci peut être le cas pour cette fonction où nous pouvons aisément nous dispenser du bloc d'instructions **`else`**. Effectivement, testez le code suivant, vous verrez qu'il fait exactement la même chose :

Code : Actionscript

```
function max(nombre1:int, nombre2:int):int
```

```
{
    var resultat:int = nombre1;
    if(nombre2 > nombre1)
    {
        resultat = nombre2;
    }
    return resultat;
}
var nombre1:int = 2;
var nombre2:int = 3;
trace("Le maximum entre " + nombre1 + " et " + nombre2 + " est " +
max(nombre1,nombre2)); // Affiche : Le maximum entre 2 et 3 est 3
```

Vous verrez que la recherche d'un optimum *(maximum ou minimum)* est une pratique assez courante avec les tableaux, mais nous verrons ceci dans le prochain chapitre !

En résumé

- Les **fonctions** permettent d'organiser le code, de le rendre plus lisible mais également de se séparer de tâches répétitives.
- Pour déclarer une fonction, on utilise le mot-clé **`function`**.
- Une fonction peut recevoir plusieurs **paramètres** en entrée, mais ne peut retourner qu'une valeur au plus.
- On utilise le mot-clé **`return`** pour renvoyer une valeur.
- On préfère généralement les **instructions de fonction** auxquelles on peut attribuer un nom.

Les tableaux

Pour clore cette première partie du cours, nous verrons les tableaux très utilisés en programmation !

Les variables de base, présentées dans un chapitre précédent, sont très utiles mais possèdent leurs limites lorsqu'il s'agit de gérer beaucoup de valeurs. Effectivement, à l'aide de variables nous pouvons associer une valeur *à un nom*. Or ceci peut être handicapant lors du traitement d'une grosse quantité de valeurs. C'est ici qu'entrent en jeu les tableaux !
Le principe des tableaux est de pouvoir stocker *plusieurs valeurs sous un même nom*. Ceci facilitera alors le traitement de ces données, puisque celles-ci seront enregistrées sous le même nom simplement différenciées par un indice à l'intérieur du tableau.

En Actionscript, nous avons plusieurs manières de créer et d'utiliser des tableaux. Dans ce chapitre nous verrons les deux types de tableaux : `Array` et `Vector`.

Le type Array

Le premier type de tableaux utilisé en Actionscript est `Array`. Ce qu'il faut savoir en Actionscript, c'est que les tableaux ne sont pas fixés en taille, ce qui permet une programmation plus souple. D'autre part en ce qui concerne le type `Array`, les tableaux ne sont pas typés, c'est-à-dire qu'ils ne sont pas réservés à un seul type de variable. Il est alors possible d'insérer dans un tableau des variables de type `int`, `uint`, `Number`, `String` ou encore tout ce qu'il vous passe par la tête.

Création

La création d'un tableau de type `Array` peut être réalisée de multiples façons. Cependant, les tableaux sont soumis aux mêmes règles que les variables. C'est pourquoi nous retrouverons donc la structure de base commune à toute variable :

Code : Actionscript

```
var nomDuTableau:Array = ceQueJeMetsDedans;
```

C'est en revanche lorsqu'il s'agit d'initialiser un tableau que nous trouvons diverses méthodes. Voici quatre manières différentes d'initialiser une variable de type `Array` :

Code : Actionscript

```
var monTableau:Array = new Array(3); // Création d'un tableau de 3 valeurs non renseignées
var monTableau:Array = new Array(); // Création d'un tableau vide
var monTableau:Array = new Array("Site", "du", "Zéro"); // Création d'un tableau contenant les 3 valeurs indiquées
var monTableau:Array = ["Site", "du", "Zéro"]; // Création du même tableau contenant les 3 valeurs indiquées
```

Dans beaucoup de langages, la longueur d'un tableau doit être définie lors de sa déclaration. Ceci n'est pas le cas en Actionscript, il est tout à fait possible de changer la longueur d'un tableau en cours de programme.

La fonction `trace()` permet également d'afficher le contenu d'un tableau. Vous pouvez ainsi à tout moment connaître l'état de votre tableau :

Code : Actionscript

```
var monTableau:Array = ["Site", "du", "Zéro"];
trace(monTableau); // Affiche : Site,du,Zéro
```

Les éléments du tableau

Même si un tableau regroupe plusieurs valeurs, celles-ci doivent pouvoir être utilisées séparément. C'est ici qu'entre en jeu la notion d'**indice** à l'intérieur d'un tableau. Ces indices sont utilisés entre crochets `[]` pour préciser la position de l'élément désiré dans le tableau. Les valeurs peuvent alors être traitées comme n'importe quelle variable :

Code : Actionscript

```
var monTableau:Array = new Array(3);
monTableau[0] = 4;
monTableau[1] = 5;
monTableau[2] = monTableau[0] + monTableau[1];
trace(monTableau[2]); // Affiche : 9
```

Attention, en programmation les indices des tableaux commencent *toujours à* `0`. Ainsi le troisième élément du tableau se trouve à l'*indice* `2`. Ce n'est pas très compliqué à comprendre, mais on a souvent tendance à l'oublier les premières fois.

Comme nous venons de le voir, les valeurs à l'intérieur d'un tableau se manipulent de la même manière que les variables. Toutefois, une valeur non initialisée dans le tableau aura une valeur par défaut, **`undefined`** :

Code : Actionscript

```
var monTableau:Array = new Array();
trace(monTableau[0]); // Affiche : undefined
```

Propriétés du type `Array`

Enfin pour en finir avec le type `Array`, nous allons voir quelques *méthodes* et *propriétés* qui pourront vous être utiles.
Tout d'abord lorsque vous utiliserez des tableaux, vous n'aurez pas besoin de connaître en permanence la taille de votre tableau. En revanche, vous pouvez à tout moment avoir besoin de connaître cette longueur de tableau. Pour cela, vous pouvez utiliser la propriété `length` associée aux tableaux de type `Array`. Voici comment l'utiliser :

Code : Actionscript

```
var monTableau:Array = new Array(5);
var taille:int = monTableau.length;
trace(taille); // Affiche : 5
```

Nous allons à présent voir différentes méthodes permettant d'insérer ou de supprimer des éléments dans un tableau.
D'abord, la méthode `push()` permet d'ajouter un ou plusieurs éléments à la fin du tableau. À l'opposé, la méthode `unshift()` insère un ou plusieurs éléments au début du tableau, c'est-à-dire à l'indice 0. Pour finir, la méthode `splice()` est un compromis car elle permet d'insérer des éléments à un indice spécifié.
Voici un exemple d'utilisation de ces méthodes :

Code : Actionscript

```
var monTableau:Array = ["Site", "du", "Zéro"];
monTableau.unshift("Bienvenue");
monTableau.splice(1, 0, "sur","le");
monTableau.push("!");
trace(monTableau); // Affiche : Bienvenue,sur,le,Site,du,Zéro,!
```

La méthode `splice()` peut avoir des finalités différentes suivant les paramètres renseignés. Voici donc la signification de chacun de ses paramètres :

- indice de départ où on se place pour effectuer les opérations
- nombre d'éléments devant être supprimés à partir de l'indice de départ
- série d'éléments qui doivent être insérés à la suite de l'indice de départ.

Ainsi la méthode `splice()` permet à la fois d'insérer des éléments mais également de supprimer ou remplacer des éléments. Dans le cas présent, la mise à zéro du deuxième paramètre permet uniquement d'insérer des éléments sans en supprimer.

Enfin, pour supprimer des éléments nous avons également trois méthodes : `pop()`, `shift()`, et `splice()`. Symétriquement à `push()` et `unshift()`, les méthodes `pop()` et `shift()` permettent de supprimer des éléments respectivement à la fin ou au début d'un tableau. Pour supprimer des éléments au milieu du tableau, il faut utiliser la méthode `splice()` présentée juste avant.
Encore une fois, voici un exemple pour bien comprendre :

Code : Actionscript

```
var monTableau:Array = ["Bienvenue", "sur", "le", "Site", "du",
"Zéro", "!"];
monTableau.pop();
monTableau.splice(1, 2);
monTableau.shift();
trace(monTableau); // Affiche : Site,du,Zéro
```

Le type Vector

Le second type de tableaux utilisé en Actionscript est `Vector`. Vous verrez que ces tableaux sont très proches de ceux de type `Array`, notamment ils sont également non fixés en taille. La principale différence vient du fait que les tableaux de type `Vector` sont *typés*. C'est-à-dire qu'une fois déclaré pour un certain type de variables, il n'est pas possible d'y mettre autre chose.
Voyons tout ceci plus en détails !

Déclaration

Ici encore la déclaration ressemble à celle d'une variable, mais à une nuance près. Étant donné que les tableaux de type `Vector` sont typés, il est nécessaire de préciser le type de variables qui sera utilisé à l'aide des chevrons « <> ». Voici donc comment déclarer un tel tableau :

Code : Actionscript

```
var nomDuTableau:Vector.<Type> = ceQueJeMetsDedans;
```

Pour initialiser un tableau de type `Vector`, voici les méthodes proposées :

Code : Actionscript

```
var monTableau:Vector.<String> = new Vector.<String>();
var monTableau:Vector.<int> = new Vector.<int>(3);
var monTableau:Vector.<String> = Vector.<String>(["Site", "du",
"Zéro"]);
```

Étant donné que ces tableaux sont typés, vous ne pouvez pas insérer une valeur dont le type ne correspondrait pas à celui déclaré. En effet, ceci entrainerait des erreurs d'exécution ou de compilation.

Notez que la taille du tableau peut toutefois être fixée en utilisant un paramètre supplémentaire de type `Boolean` et valant **`true`**. Voici le code correspondant :

Code : Actionscript

```
var monTableau:Vector.<int> = new Vector.<int>(3, true);
```

Gestion des éléments

Tout comme les tableaux `Array`, les valeurs internes peuvent être manipulées à l'aide d'indices mis entre crochets « `[]` ». Voici encore quelques exemples de manipulations de valeurs à l'intérieur d'un tableau :

Code : Actionscript

```
var monTableau:Vector.<int> = new Vector.<int>(10);
monTableau[0] = 0;
monTableau[1] = 1;
for(var i:int = 2; i < monTableau.length; i++)
{
    monTableau[i] = monTableau[i-1] + monTableau[i-2];
}
trace(monTableau); // Affiche : 0,1,1,2,3,5,8,13,21,34
```

Pour les plus perspicaces, vous aurez vu apparaître les premiers termes de la *Suite de Fibonacci* qui sont en effet : 0,1,1,2,3,5,8,13,21,34.
En augmentant la taille du tableau, vous pourrez ainsi avoir tous les termes de la suite que vous désirez !

Grâce à cet exemple nous commençons à cerner l'intérêt de l'utilisation des tableaux. Effectivement lorsque nous utilisons des boucles par exemple, il devient relativement simple de lier notre variable `i` de boucle avec les indices du tableau. Ceci avait déjà été dit précédemment mais nous allons le rappeler, les boucles sont extrêmement utiles et performantes pour parcourir des tableaux.

Pour allier tableaux et fonctions, nous allons transformer le code précédent pour créer une fonction renvoyant le énième terme de la suite de Fibonacci. Découvrons cette fonction :

Code : Actionscript

```
function suiteFibonacci(terme:int):uint
{
    var monTableau:Vector.<uint> = new Vector.<uint>(terme);
    monTableau[0] = 0;
    monTableau[1] = 1;
    for(var i:int = 2; i < monTableau.length; i++)
    {
        monTableau[i] = monTableau[i-1] + monTableau[i-2];
    }
    return (monTableau[terme-1]);
}
trace(suiteFibonacci(10)); // Affiche : 34
trace(suiteFibonacci(20)); // Affiche : 4181
trace(suiteFibonacci(30)); // Affiche : 514229
```

Enfin pour finir, vous noterez que les méthodes `push()`, `pop()`, `shift`, `unshift()` et `splice()` fonctionnent également avec les tableaux de type `Vector`.

Les tableaux multidimensionnels

Le concept

Nous allons maintenant découvrir les tableaux **multidimensionnels** : un nom barbare pourtant c'est une notion qui n'est pas si compliquée que ça !
Les tableaux multidimensionnels ne sont en réalité que des tableaux imbriqués dans d'autres tableaux. Certes c'est une notion qui fait peur aux premiers abords, cependant il n'y a rien de nouveau en matière de code.
Pour mieux comprendre, nous allons illustrer tout ça d'un exemple. Nous pourrions créer une liste regroupant l'ensemble des livres de la collection « le Livre du Zéro », il serait alors possible de réaliser cela avec un tableau classique ou monodimensionnel. Imaginons maintenant qu'en plus de stocker le nom des livres, nous souhaitions y ajouter le nom de l'auteur et l'année de sortie. Dans ce cas nous avons besoin d'utiliser un tableau **bidimensionnel** de la manière suivante :

Code : Actionscript

```
var livres:Array = new Array();
livres[0] = ["Réalisez votre site web avec HTML5 et CSS3", "Mathieu
Nebra", 2011];
livres[1] = ["Apprenez à programmer en Java", "Cyrille Herby",
2011];
```

```
livres[2] = ["Débutez dans la 3D avec Blender", "Antoine Veyrat",
2012];
livres[3] = ["Rédigez des documents de qualité avec LaTeX", "Noël-
Arnaud Maguis", 2010];
```

Sachant que les éléments du tableau principal sont des tableaux, vous obtiendrez donc un tableau en utilisant la notation avec indice vue précédemment :

Code : Actionscript

```
trace(livres[0]); // Affiche : Réalisez votre site web avec HTML5 et
CSS3,Mathieu Nebra,2011
```

Il est bien évidemment possible d'accéder à une valeur à l'intérieur du second tableau. Pour cela nous devrons utiliser une notation à *double indice* ; c'est ce qui rend le tableau multidimensionnel !
Regardez plutôt ce code :

Code : Actionscript

```
trace(livres[0][0]); // Affiche : Réalisez votre site web avec HTML5
et CSS3
```

Pour bien comprendre, nous pouvons effectuer une analogie avec un tableau Excel par exemple où le premier indice correspondrait au numéro de la ligne et le second au numéro de la colonne.
Pour vos besoins, sachez que vous pouvez réaliser des tableaux de la dimension que vous souhaitez.

Un peu de pratique

Dans cette section, nous allons réaliser plusieurs fonctions pour apprendre à bien manipuler les tableaux. Pour les tests nous reprendrons le tableau précédent :

Code : Actionscript

```
var livres:Array = new Array();
livres[0] = ["Réalisez votre site web avec HTML5 et CSS3", "Mathieu
Nebra", 2011];
livres[1] = ["Apprenez à programmer en Java", "Cyrille Herby",
2011];
livres[2] = ["Débutez dans la 3D avec Blender", "Antoine Veyrat",
2012];
livres[3] = ["Rédigez des documents de qualité avec LaTeX", "Noël-
Arnaud Maguis", 2010];
```

Pour éviter de surcharger le code de ce cours, nous ne réécririons pas ces lignes avant chaque fonction. À vous de les copier si vous souhaitez réaliser des essais.

Qui est-ce ?

La fonction présentée ci-dessous permet de retrouver l'auteur d'un livre. Pour cela nous réaliserons donc une boucle à l'intérieur de laquelle nous rechercherons le livre spécifié en paramètre. Une fois le livre trouvé, il ne reste plus qu'à retenir l'indice correspondant dans le tableau et de renvoyer le nom de l'auteur.
Voici la fonction en question :

Code : Actionscript

```
function auteur(monTableau:Array, monLivre:String):String
{
```

```
        var i:int = 0;
        var continuer:Boolean = true;
        do
        {
            if(monTableau[i][0] == monLivre)
                continuer = false;
            i++;
        } while(continuer);
        return monTableau[i-1][1]
    }
    trace(auteur(livres, "Apprenez à programmer en Java")); // Affiche :
    Cyrille Herby
    trace(auteur(livres, "Réalisez votre site web avec HTML5 et CSS3"));
    // Affiche : Mathieu Nebra
```

Vous remarquerez que dans cet exemple, nous combinons à la fois : condition, boucle, fonction et tableau. Si vous avez bien compris cet exemple c'est que vous êtes au point sur la première partie et donc fin prêt pour entamer la deuxième ! 

Trions par chronologie

Comme cela est dit dans le titre, nous allons dans cette fonction trier le tableau en fonction de l'année de sortie des livres. Pour faire ceci, nous allons devoir créer un nouveau tableau que nous allons remplir au fur et à mesure. Nous prendrons donc chaque livre du premier tableau, puis nous chercherons où l'insérer dans le second. Pour réaliser cela, nous devrons utiliser deux boucles, une pour chaque tableau.
Cette fonction est plus complexe que la précédente :

Code : Actionscript

```
    function tri(monTableau:Array):Array
    {
        var nouveauTableau:Array = new Array();
        nouveauTableau[0] = monTableau[0];
        for(var i:int = 1; i < monTableau.length; i++)
        {
            var j:int = 0;
            var continuer:Boolean = true;
            while(continuer)
            {
                if(j >= nouveauTableau.length || monTableau[i][2] <=
    nouveauTableau[j][2])
                    continuer = false;
                j++;
            }
            nouveauTableau.splice(j-1, 0, monTableau[i]);
        }
        return nouveauTableau
    }
    livres = tri(livres);
    trace(livres[0]); // Affiche : Rédigez des documents de qualité avec
    LaTeX,Noël-Arnaud Maguis,2010
    trace(livres[1]); // Affiche : Apprenez à programmer en
    Java,Cyrille Herby,2011
    trace(livres[2]); // Affiche : Réalisez votre site web avec HTML5 et
    CSS3,Mathieu Nebra,2011
    trace(livres[3]); // Affiche : Débutez dans la 3D avec
    Blender,Antoine Veyrat,2012
```

Prenez le temps de bien comprendre ces deux exemples qui reprennent la quasi-totalité des concepts de la première partie.

Parcourir un tableau

Il existe plusieurs manières de parcourir un tableau, que ce soit un `Array` ou un `Vector`. Il faut savoir que certaines sont plus rapides que d'autres, et il faut savoir choisir laquelle utiliser en fonction des possibilités. Nous utiliserons à chaque fois la boucle **`for`** que nous avons déjà vue précédemment, néanmoins l'Actionscript en propose quelques « variantes » encore plus efficaces.

La boucle for classique

Comme nous l'avons vu, la boucle **for** est particulièrement bien adaptée au parcours d'un tableau.
Pour rappel, voici un exemple de manipulation de ce type de boucle :

Code : Actionscript

```
var monTableau:Array = ["élément n°1", "élément n°2", "élément n°3",
"élément n°4"];
for(var i:int = 0; i < monTableau.length; i++)
{
    trace(monTableau[i]);
}
```

Ce qui affichera :

Code : Console

```
élément n°1
élément n°2
élément n°3
élément n°4
```

Dans une boucle **for** classique, nous nous basons donc sur l'indice, ou la position, des différents éléments à l'intérieur du tableau. Pour cela, nous avons alors besoin de connaître la longueur de celui-ci, que nous pouvons d'ailleurs stocker dans une variable :

Code : Actionscript

```
var monTableau:Array = ["élément n°1", "élément n°2", "élément n°3",
"élément n°4"];
var longueur:int = monTableau.length;
for(var i:int = 0; i < longueur; i++)
{
    trace(monTableau[i]);
}
```

Cependant il est également possible d'utiliser d'autres boucles, où la connaissance de la longueur d'un tableau devient superflue ; j'ai nommé les boucles **for...in** et **for each** !

La boucle for...in

La boucle **for...in** est très proche de la boucle **for** classique, et s'utilise quasiment de la même manière. Toutefois ici, il n'est pas nécessaire de connaître le nombre d'éléments contenus dans un tableau pour parcourir l'intégralité de ce dernier.
Voici un exemple d'utilisation de cette boucle :

Code : Actionscript

```
var monTableau:Array = ["élément n°1", "élément n°2", "élément n°3",
"élément n°4"];
for(var i:String in monTableau)
{
    trace(monTableau[i]);
}
```

Ce qui nous affichera comme précédemment :

Code : Console

```
élément n°1
élément n°2
élément n°3
élément n°4
```

Ainsi vous pouvez considérer que dans une boucle **for...in** la variable se comporte comme un indice qui s'incrémente automatiquement et qui parcourt l'intégralité du tableau.

Il est fort probable que certains d'entre vous se demandent alors, pourquoi diable, la variable `i` est-elle de type `String` et non de type `int` ?

Pour comprendre cela, nous allons devoir brièvement sortir du cadre de ce chapitre. En réalité la boucle **for...in** n'est pas propre aux tableaux, mais peut être utilisée pour parcourir d'autres types d'éléments ou objets...
Pour cerner le problème par un exemple, nous allons imaginer l'objet ou la variable suivante :

Code : Actionscript

```actionscript
var monObjet:Object = {a:"élément A", b:"élément B", c:"élément C"};
```

Nous avons donc ici une sorte de tableau contenant trois éléments. Néanmoins, ces éléments ne sont pas rangés à l'aide d'un indice, mais plutôt grâce à une *étiquette* ou *propriété* et ne sont donc pas disposés suivant un ordre particulier. La seule manière de faire référence à un élément à l'intérieur de cette variable est donc d'utiliser les *étiquettes* associées à chacun de ces éléments.

L'utilisation de la boucle **for...in** prend alors tout son sens :

Code : Actionscript

```actionscript
for(var i:String in monObjet)
{
    trace(i + ": " + monObjet[i]);
}
```

Voici le résultat :

Code : Console

```
a: élément A
b: élément B
c: élément C
```

Pour utiliser correctement la boucle **for...in**, vous devez vous rappeler qu'à l'intérieur de celle-ci nous travaillons non pas à l'aide d'un indice numérique mais bien avec une étiquette sous forme de chaîne de caractères. Dans le cas d'un tableau de type `Array` ou `Vector`, la variable utilisée représente néanmoins l'indice, mais toujours sous la forme d'une chaîne de caractères de type `String` !

La boucle for each

La dernière boucle permettant de parcourir un tableau est la boucle **for each**. Contrairement à ses congénères, la variable utilisée à l'intérieur de celle-ci ne fait pas référence à l'indice, ou l'étiquette, associé aux éléments d'un tableau, mais aux éléments eux-mêmes.

La manipulation des éléments devient alors beaucoup plus simple :

Code : Actionscript

```
var monTableau:Array = [12, 42, 10, 3];
for each(var nombre:int in monTableau)
{
    trace(nombre);
}
```

Comme vous l'imaginez, dans ce cas, le type de la variable devra s'adapter suivant la nature des éléments contenus à l'intérieur du tableau :

Code : Actionscript

```
var monTableau:Array = [2.3, 24567673, 42, 3.14, 276.54];
for each(var nombre:Number in monTableau)
{
    trace(nombre);
}
```

Ou encore :

Code : Actionscript

```
var monTableau:Array = ['Hello', 'world', '!'];
for each(var chaine:String in monTableau)
{
    trace(chaine);
}
```

Notez toutefois qu'en utilisant cette boucle vous perdez la notion de repérage à l'intérieur du tableau, et vous n'êtes alors plus en mesure de relier l'élément au tableau.

Bien évidemment ce qui est faisable avec un tableau de type `Array`, l'est aussi avec un autre tableau de type `Vector` :

Code : Actionscript

```
var monVecteur:Vector.<String> = Vector.<String>(['Hello', 'world',
'!']);
for each(var chaine:String in monVecteur)
{
    trace(chaine);
}
```

En résumé

- Grâce aux **tableaux**, nous pouvons stocker plusieurs valeurs sous un même nom de variable.
- Les éléments d'un tableau sont accessibles via un **indice** qui démarre à `0`.
- Les tableaux de type `Array` ne sont ni fixés, ni typés.
- Les tableaux de type `Vector` ne sont pas fixés, mais en revanche ils sont typés.
- Différentes méthodes permettent de manipuler des tableaux à la fois de `Array` et de type `Vector`.
- On utilise généralement les boucles **`for`**, **`for...in`** ou **`for each`** pour parcourir un tableau.
- Il est possible de créer des tableaux **multidimensionnels** en utilisant une notation à double indice.

Partie 2 : La programmation orientée objet

Maintenant que vous êtes au point sur les bases du langage, il est temps d'affronter le gros morceau de ce cours : la Programmation Orientée Objet.
C'est une partie très importante pour programmer en Actionscript 3, donc prenez votre temps et n'hésitez pas à la relire ! 

La POO dans tous ses états

Nous allons maintenant attaquer la deuxième partie du cours, traitant d'une notion très importante en Actionscript 3 : la **Programmation Orientée Objet** (ou POO).
Appréhender cette notion sera peut-être difficile au début, mais vous finirez par tomber sous le charme de cette façon de programmer, c'est promis !

Cette partie sera également l'occasion de faire (enfin) un peu de pratique à l'occasion de quelques exercices que nous réaliserons ensemble.

Attachez vos ceintures, c'est parti ! La POO n'aura bientôt plus aucun secret pour vous ! 

Les notions-clés

Il était une fois... un objet

Vous avez sûrement une idée de ce qu'est un objet : dans la vie de tous les jours, nous en voyons, nous en touchons des centaines. Ils ont chacun une forme et remplissent une ou plusieurs tâches. On distingue rapidement plusieurs catégories d'objets qui se ressemblent : quand vous mettez la table, vous manipulez des objets (fourchettes, couteaux, cuillères, ...) appartenant à la catégorie « couvert ». Bien sûr, vous pouvez avoir plusieurs objets de cette catégorie, voire plusieurs fourchettes identiques. Les objets de la catégorie « fourchette » font également partie de la catégorie « couvert », car les fourchettes ne sont en fait que des couverts particuliers.

La programmation orientée objet part de ces différentes observations pour introduire de nouveaux concepts en programmation, afin de rendre le développement de logiciels plus efficace et intuitif.

L'autre façon de programmer est appelée le **procédural**, où il n'y a grosso-modo que des variables, des fonctions et des instructions effectuées les unes à la suite des autres. La POO est différente par bien des aspects.

L'Objet

Prenons une de vos fourchettes.

Une fourchette

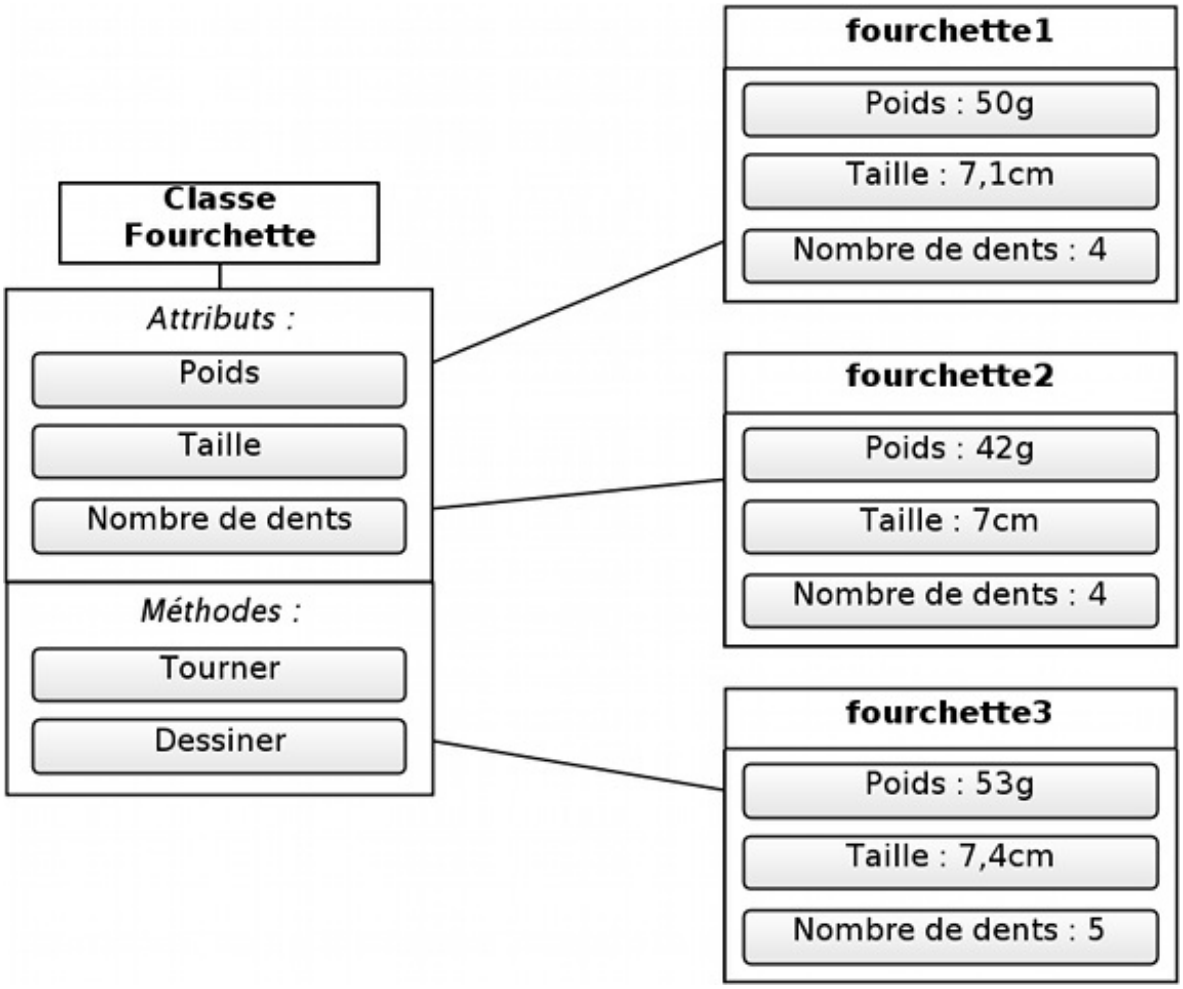
Il s'agit d'un **objet**, ayant une forme particulière et une certaine utilité, que l'on peut retrouver dans les autres fourchettes. Un objet en programmation est également une entité possédant des propriétés que l'on peut retrouver sur d'autres objets de la même catégorie. Il ne s'agit ni d'une variable, ni d'une fonction, mais d'un mélange des deux : c'est un nouveau concept à part entière. Un objet peut donc contenir plusieurs variables (appelées **attributs**) et plusieurs fonctions (appelées **méthodes**). Ces variables et fonctions (appelées **propriétés**) sont profondément liées à l'objet qui les contient : les attributs stockent des informations sur cet objet. Par exemple, si mon objet « fourchette » possède un attribut « poids », cet attribut représente la masse de cette fourchette, et non la masse d'une autre fourchette posée à côté.

La Classe

Seulement, vous avez sur votre table non pas une mais dix fourchettes, quoique pas exactement identiques. En effet, cette fourchette-ci et plus légère que cette fourchette-là, donc son attribut « poids » est plus faible. Mais ces deux objets restent des fourchettes, malgré leur différence de poids. `Fourchette` est donc la catégorie de ces objets, appelée **classe**.

On dit que les objets d'une classe sont des **instances** (ou **occurrences**) de cette classe.

Une classe décrit la nature de ses objets : c'est un *schéma* permettant de fabriquer et de manipuler plusieurs objets possédant les mêmes attributs et méthodes. Par exemple, la classe `Fourchette` pourrait décrire les attributs « poids », « taille », « nombre de dents », etc.



La classe Fourchette et quelques instances

Nous pourrions aussi créer une classe `Couteau` pour les couteaux, et une classe `Cuillère` pour les cuillères.

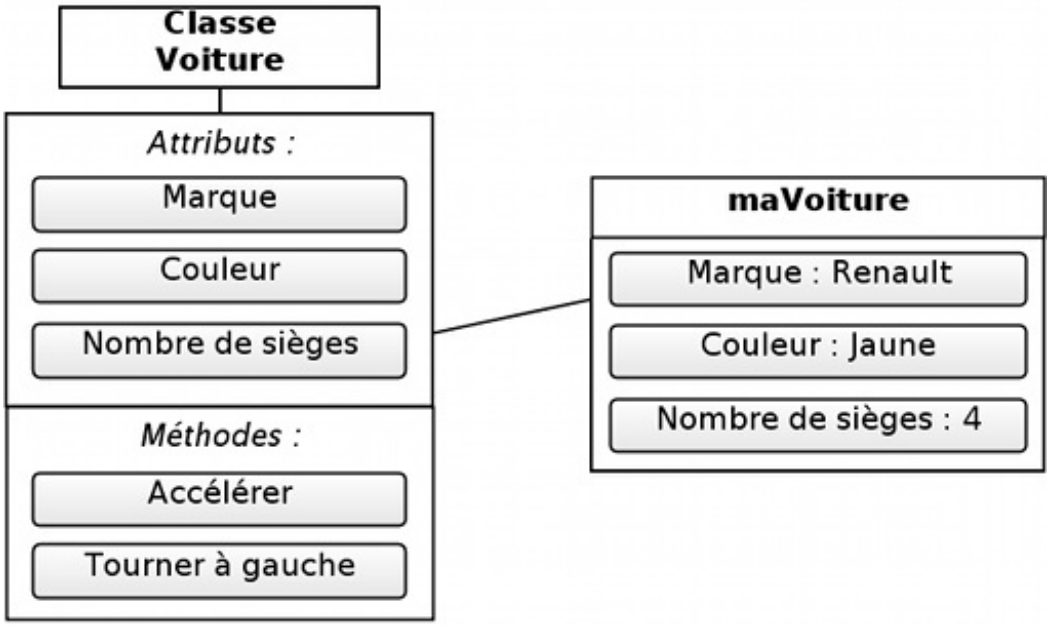
En Actionscript 3, les classes sont considérées comme des types, au même titre que les `int`, `Number` ou `String`.

Un autre exemple

Une fois la table mise, vous tournez la tête et admirez à travers la fenêtre votre magnifique voiture toute neuve : ne serait-ce pas un objet de la classe `Voiture` ? Avec un objet aussi complexe, cette classe contient certainement plus d'attributs et de méthodes que la classe `Fourchette`...

Votre magnifique voiture

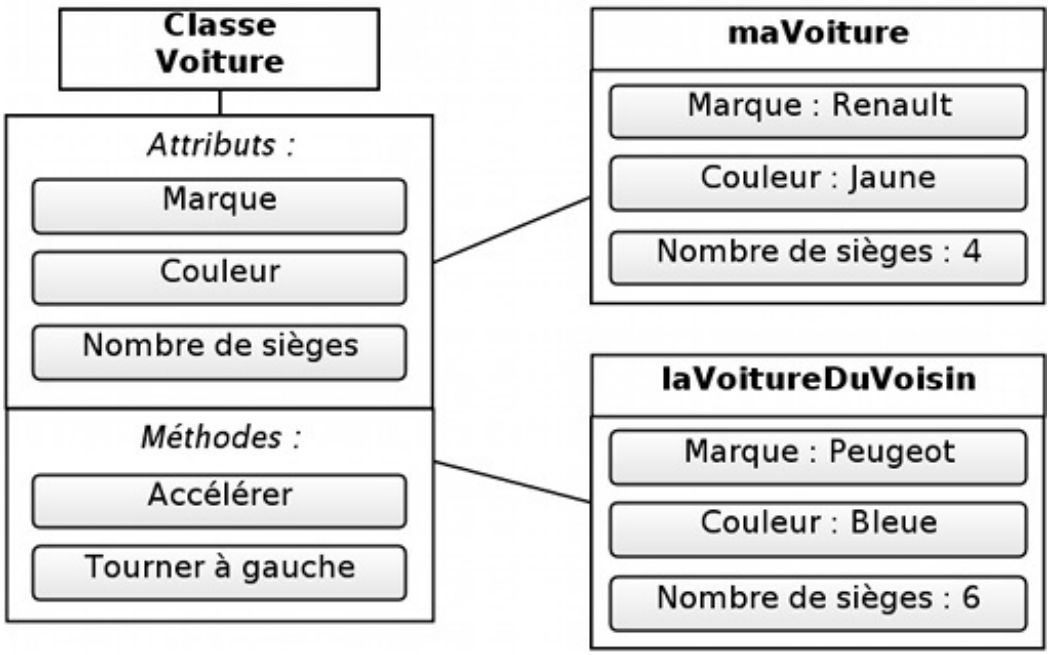
Voici quelques exemples d'attributs : « taille » (un grand classique), « marque », « couleur », « nombre de pneus », « nombre de sièges » ; et quelques exemples de méthodes : « verrouiller les portières », « démarrer le moteur », « accélérer », « freiner », « tourner à gauche », « tourner à droite »...



La classe Voiture et une instance

Vous apercevez un peu plus loin la voiture de votre voisin, garée en face. C'est également un objet de la classe `Voiture`, avec une marque, une couleur, un nombre de pneus et de sièges, la possibilité d'accélérer, de freiner, de tourner à gauche ou à droite... Seulement elle n'a ni la même marque, ni la même couleur que la vôtre.

Ces deux objets sont des instances de la classe `Voiture`.



La voiture de mon voisin est une autre instance

L'encapsulation

Vous vous souvenez avoir ouvert le capot de votre voiture après l'avoir garé : vous avez intensément regardé ce qu'il y a à l'intérieur. Malgré votre immense effort de concentration, le fonctionnement du moteur, de tous ces câbles, circuits électriques et autres tuyaux vous échappe encore...

Sous le capot...

Fort heureusement, lorsque vous refermez le capot et montez dans votre voiture, vous oubliez toute cette mécanique trop compliquée, et vous conduisez à l'aide d'un volant, de pédales et d'un levier de boîte de vitesse. Grâce à eux, vous pouvez utiliser votre voiture sans être ingénieur en génie mécanique ou électronique !

Votre volant vous facilite bien la tâche...

Imaginez maintenant que vous écriviez une classe très complexe, qui permet de faire plein de choses incroyables ! Malheureusement, personne ne voudra de votre magnifique classe : qui serait assez fou pour se plonger dans votre code et tentez de le déchiffrer ?

Ce à quoi votre code pourrait ressembler...

En effet, votre code est un peu comme le bazar qui se trouve sous le capot de votre voiture. Mais vous êtes malins, et vous fournissez aux autres développeurs des méthodes très simples et faciles à comprendre, tout en cachant littéralement sous le capot ce qu'elles font, à l'instar du volant, des pédales et du frein à main. Ainsi, les programmeurs qui utiliseront votre classe (y compris vous-mêmes) auront un volant à disposition au lieu d'un moteur et vingt-et-un tuyaux. On appelle ce principe l'**encapsulation**.

L'héritage

Puis, votre regard se porte sur l'autoroute au loin : vous distinguez d'autres voitures. Mais des formes différentes se déplacent parmi elles : ce ne sont de toute évidence pas des voitures, mais plutôt des camions et des motos. Vous remarquez que ces objets comportent eux aussi des roues ; ils ont eux aussi une couleur et une marque ; ils peuvent eux aussi accélérer, freiner, tourner à gauche, tourner à droite... Ce seraient presque des objets de la classe `Voiture`.

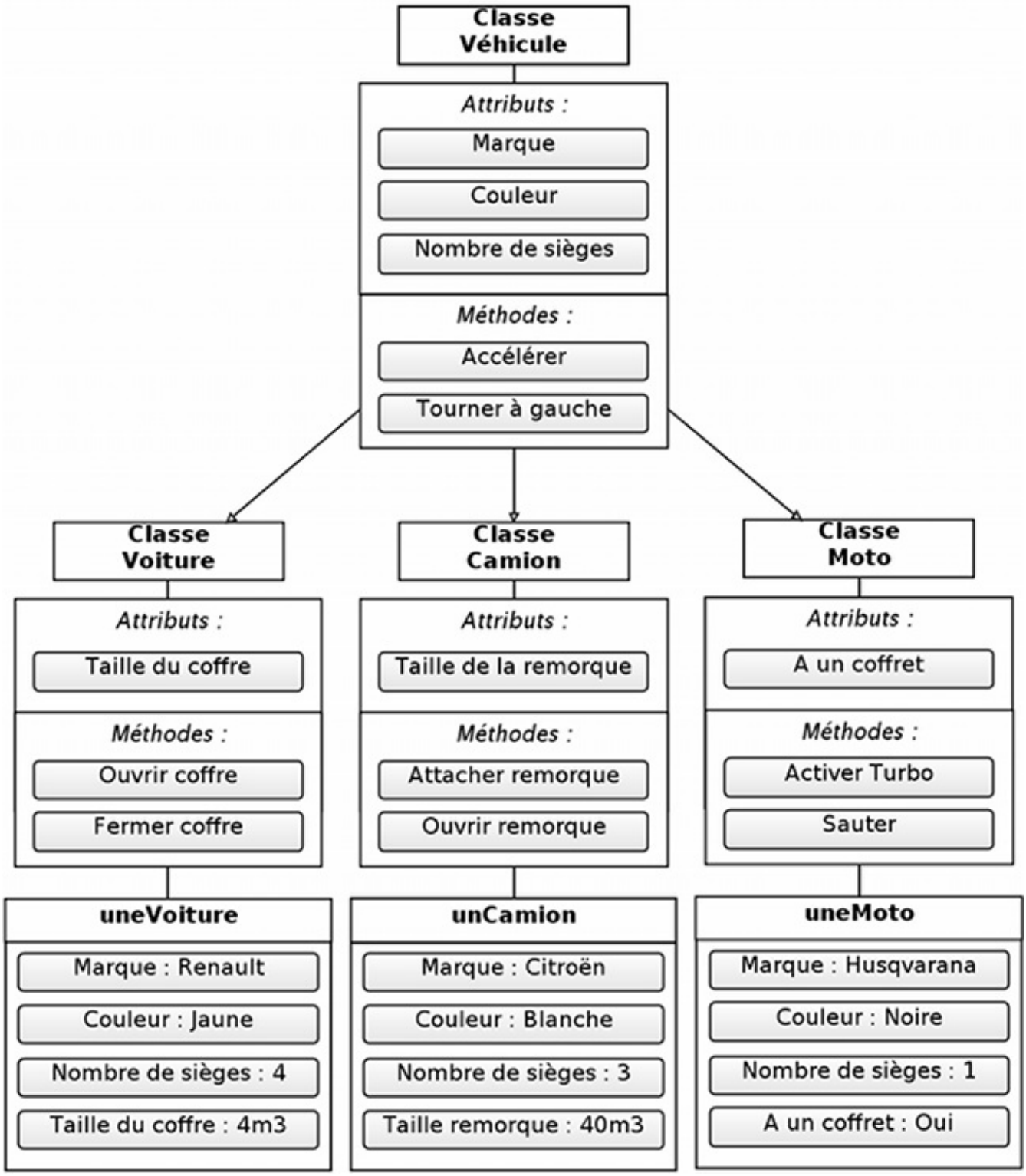
Une autoroute belge

Mince alors ! Il va falloir refaire de nouvelles classes quasiment identiques...
Pourquoi ne pas faire une classe `Véhicule` plus généraliste, qui décrirait les attributs et méthodes communs aux classes `Voiture`, `Camion` et `Moto` ? C'est une excellente idée !

Notre nouvelle classe `Véhicule` décrit donc les attributs et les méthodes communs à tous les véhicules routiers, comme le nombre de roues et de sièges, la marque, la couleur, les méthodes « accélérer », « freiner », « tourner à gauche », « tourner à droite »... On l'appelle la **classe mère** (ou **superclasse**), par opposition aux classes `Voiture`, `Camion` et `Moto` qui sont ses **classes filles** (ou **sous-classes**). On dit également que les classes `Voiture`, `Camion` et `Moto` héritent de la classe `Véhicule`.

Désormais, inutile de réécrire les attributs et méthodes de la classe `Véhicule` dans la classe `Voiture`, car cette dernière hérite des attributs et des méthodes de la classe `Véhicule` !

Ainsi tous nos objets de la classe `Voiture` sont également des objets de la classe `Véhicule`, mais leur description dans la classe `Voiture` est plus précise (par exemple, les voitures ont un coffre, alors que les camions ont une remorque).



Les trois classes sont des sous-classes de la classe Véhicule

L'un des intérêts de l'héritage est donc de pouvoir créer plusieurs classes différentes dérivant d'une classe mère, sans avoir à recopier les attributs et les méthodes communes dans chaque classe. C'est aussi une façon de penser et concevoir le programme de manière logique et cohérente.

Si nous revenions sur le premier exemple, la classe `Fourchette`, ainsi que les classes `Couteau` et `Cuillère`, sont des sous-classes de la classe `Couvert` !

Manipuler des objets : les chaînes de caractères

L'horrible secret du type `String`

Vous souvenez-vous des chaînes de caractères que nous avons vu dans la première partie ?

Code : Actionscript

```
var coucou:String = "Hello world!";
```

Regardez bien le type de la variable : `String`... Il cache un terrible secret : il s'agit en réalité d'une *classe* !
Et oui ! En écrivant la ligne au-dessus, nous avons sans le savoir créé une instance de la classe `String`. Et nous avons également utilisé des propriétés de cette classe ! 

Créer un objet

Tout d'abord, il nous faut apprendre à créer des objets. Il y a une syntaxe particulière à respecter :

Code : Actionscript

```
new String(); // Je créé un objet de la classe String (je l'ai
d'ailleurs créé dans le vide, cette instruction ne servira à rien
:D)
```

Comme vous pouvez le voir, nous utilisons le mot-clé **`new`**, suivi du nom de la classe de l'objet que nous voulons créer, et d'éventuels paramètres entre parenthèses (comme pour les fonctions). Par exemple, si nous voulions créer un objet de la classe `String` contenant la chaîne `"Hello world!"`, nous procéderions ainsi :

Code : Actionscript

```
var coucou:String = new String("Hello world!"); // Je créé un objet
String contenant la chaîne "Hello world!"
```

Mais, avant nous n'avions pas besoin de **`new`** pour faire ça !

Effectivement, la classe `String` possède une particularité : c'est un **type de données**. Cela signifie que les objets de cette classe sont un peu spéciaux : ils peuvent agir en tant que simple donnée (nombre, chaîne de caractères, etc.). Ainsi, nous pouvons nous passer de la syntaxe de création d'un objet :

Code : Actionscript

```
var coucou:String = "Hello world!";
```

revient à écrire :

Code : Actionscript

```
var coucou:String = new String("Hello world!");
```

Qu'en est-il de `int`, `uint` et `Number` ?

Et bien, figurez-vous que ce sont également des classes, qui fonctionnent de la même manière que la classe `String` !

Il est impossible d'utiliser le mot-clé **`new`** pour créer des objets des classes `int` et `uint` ! C'est la raison pour laquelle elles commencent par une minuscule, pour bien marquer la différence avec les autres classes.

En effet, ces deux classes sont encore plus spéciales et n'utilisent qu'une seule syntaxe : celle que vous avez apprise dans la

première partie du cours. Par contre, la classe `Number` fonctionne tout à fait normalement, comme la classe `String` :

Code : Actionscript

```
var entier:int = new int(-47); // INTERDIT ! ! !
var entier2:int = -42; // Autorisé

var entierNonSigne:uint = new uint(47); // NOOON ! Tout sauf ça !
var entierNonSigne2:uint = 47; // Ouf !

var nombre:Number = new Number(3.1415); // Pas de problème
var nombre2:Number = 3.1415; // Pour les flemmards :D

var monNom:String = new String("Bryan"); // Je sens que je connais
cette phrase...
var ouSuisJe:String = "In the kitchen!"; // Argh !
```

Accéder aux propriétés d'un objet

Reprenons un peu plus de code que nous avons écrit dans la première partie :

Code : Actionscript

```
var coucou:String = new String("Hello world!");
trace("Cette chaîne contient " + coucou.length + " caractères.");
// Affiche : Cette chaîne contient 13 caractères.
```

et observons plus particulièrement cette expression : `coucou.length`.

Nous avons utilisé la propriété `length` de la classe `String`, qui renvoie la longueur de la chaîne !
Décortiquons cette instruction :

- Tout d'abord, nous renseignons le nom de la variable contenant notre objet ; ici, il s'agit de `coucou`, variable que nous avons déclaré plus haut avec cette ligne : `var coucou:String = "Hello world !";`.
- Ensuite, nous utilisons le caractère point « `.` » pour signaler que nous allons utiliser un attribut ou une méthode de cet objet-là.
- Enfin, nous spécifions quelle propriété nous voulons utiliser : ici, nous tapons donc `length`, la propriété de la classe `String` qui nous intéresse.

Nous obtenons ainsi la longueur de l'objet « chaîne de caractères » contenu dans la variable `coucou` ! En français, cela donnerait quelque chose comme « Lis la propriété longueur de l'objet contenu dans ma variable coucou ».

Des pointeurs sous le capot
Plantons le décor

Prenons notre classe `Voiture`, et supposons qu'elle a une propriété `marque` de type `String` qui contient le nom de la marque des voitures.

Créons un objet de la classe `Voiture` et spécifions sa marque :

Code : Actionscript

```
var lienVoiture:Voiture = new Voiture(); // Je créé ma voiture...
lienVoiture.marque = "Peugeot"; // ...de marque Peugeot.
```

Déclarons une deuxième variable :

Code : Actionscript

```
var lienVoiture2:Voiture; // Je déclare une autre variable
```

La variable n'est pas initialisée, elle contient donc **null**.

Ensuite, procédons à une affectation :

Code : Actionscript

```
lienVoiture2 = lienVoiture; // C'est bien une affectation... Mais
que fait-elle réellement ?
```

Nous allons regarder de quelle marque sont les voitures (vous avez sûrement deviné) :

Code : Actionscript

```
trace(lienVoiture.marque; // Affiche : Peugeot
trace(lienVoiture2.marque); // Affiche : Peugeot
```

Bien ! Maintenant, modifions la marque de `lienVoiture2` :

Code : Actionscript

```
lienVoiture2.marque = "Renault";
// Nous pourrions également faire :
lienVoiture2.marque = new String("Renault");
```

Enfin, regardons à nouveau la marque des voitures :

Code : Actionscript

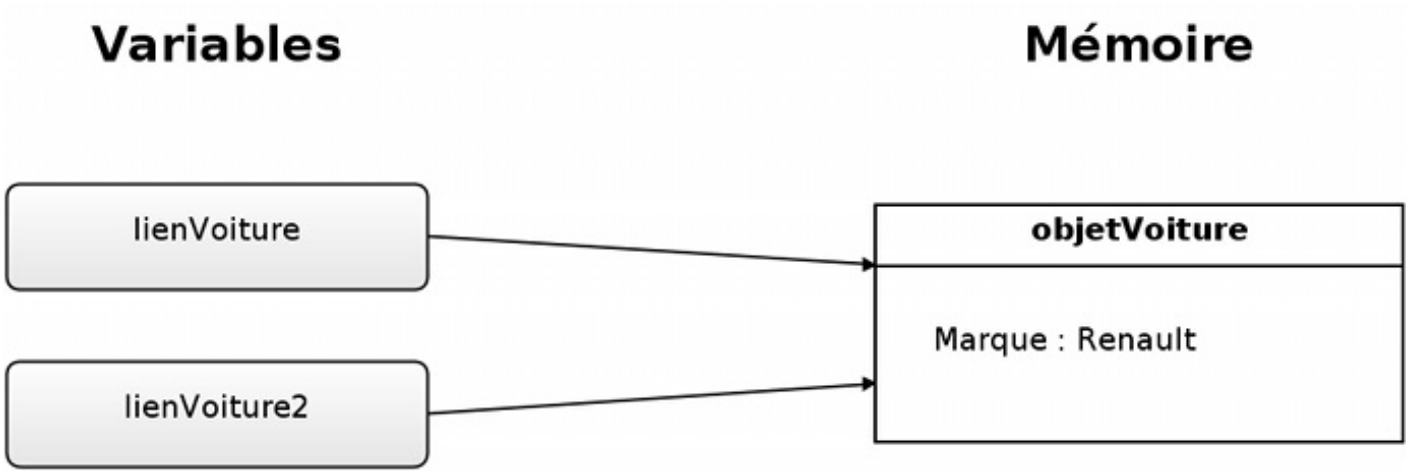
```
trace(lienVoiture.marque); // Affiche : Renault
trace(lienVoiture2.marque); // Affiche : Renault
```

Horreur ! Les deux ont la même marque !
La réponse est simple : il s'agit du même objet ! Et oui, il n'y en a qu'**un seul et unique** !

Explications

Lorsque nous créons l'objet de classe `Voiture`, il est stocké dans la mémoire vive de votre système. Ensuite, quand nous l'affectons à une variable, un lien vers cet objet est créé, puis stocké dans la variable. On dit alors que les variables sont des **pointeurs** : elles *pointent du doigt* l'objet qui leur est associé, afin que nous, programmeurs, puissions accéder à cet objet. D'ailleurs, vous aurez peut-être remarqué le nom précis que j'ai donné à mes variables : `lienVoiture` et `lienVoiture2` ; n'est-ce pas évocateur ?

Bien évidemment, j'ai nommé les variables ainsi dans un but pédagogique : évitez d'appeler vos variables de cette façon, cela ne servirait à rien (et tous vos collègues vous dévisageraient d'un air bizarre).



Les variables-pointeurs sont toutes les deux liées à l'objet.

Donc, lorsque nous écrivons notre affectation, nous nous contentons en réalité de recopier le **pointeur** vers l'objet initialement créé :

Code : Actionscript

```
lienVoiture2 = lienVoiture; // Recopions le pointeur de lienVoiture
dans lienVoiture2
```

Ainsi, en utilisant les propriétés de l'objet de `lienVoiture2`, nous utilisons également celles de l'objet de `lienVoiture`. Logique : il s'agit du même objet !

Et si nous créions un autre objet de classe `Voiture` ?

Très bonne idée ! Créons un nouvel objet et affectons-le à la variable `lienVoiture2` :

Code : Actionscript

```
lienVoiture2 = new Voiture(); // Un deuxième objet Voiture entre en
scène !
```

Modifions sa marque :

Code : Actionscript

```
lienVoiture2.marque = "Citroën";
```

Et regardons la marque des deux voitures :

Code : Actionscript

```
trace(lienVoiture.marque); // Affiche : Renault
trace(lienVoiture2.marque); // Affiche : Citroën
```

Ouf ! Quel soulagement ! Nous avons bien deux objets distincts de la classe `Voiture` !

En résumé

- Les objets contiennent des **propriétés** : les variables sont appelées **attributs** et les fonctions **méthodes**.
- Chaque objet est décrit par une **classe** : il s'agit d'une sorte de plan de construction des objets. On dit que les objets sont

des **instances** (ou **occurrences**) de leur classe.
- L'**encapsulation** est un principe important en POO qui consiste à cacher le fonctionnement interne des classes, et à montrer seulement une interface simplifiée.
- L'**héritage** est un autre principe, tout aussi important, consistant à faire hériter des classes (dites **classes filles**) d'une **classe mère**.
- En Actionscript 3, tout est objet : tout ce que vous manipulerez, ce sera des objets.
- Le mot-clé `new` permet de créer des objets.
- La plupart des variables sont en réalité des **pointeurs** (ou **liens**) vers des objets stockés en mémoire.

Les classes (1ère partie)

Dans le chapitre précédent, nous avons introduit la notion de *classe*, et nous avons même appris à utiliser la classe qu'est `String`. C'est déjà un bon début dans la découverte de la programmation orientée objet, mais ce n'est pas suffisant. À partir de maintenant, nous allons apprendre à créer nos propres classes.
En utilisant le concept d'objets ou de classes, la programmation orientée objet permet d'organiser le code vraiment différemment de la programmation procédurale.

Étant donné qu'il y a beaucoup de choses à dire sur le sujet, la théorie sur *les classes* s'étalera sur deux chapitres. Nous allons donc voir dans ce chapitre les principes fondamentaux, puis nous complèterons avec quelques notions supplémentaires dans le prochain chapitre.

Créer une classe

La Classe

Rappels

Comme nous l'avons déjà mentionné, une classe est l'ensemble du code permettant de représenter un objet dans un programme. Celle-ci est constituée de variables appelées **attributs** et de fonctions appelées **méthodes**. On parle également de **propriétés** pour définir l'ensemble des *attributs* et des *méthodes* d'une classe.
Ainsi grâce à ces propriétés, nous allons pouvoir structurer notre objet, et définir l'intégralité de ses caractéristiques.

Une fois que la classe est définie, nous pouvons alors créer des **instances** ou **occurrences** de cette classe. Pour vous faire une idée, on peut considérer une *instance* comme un objet « physique », en opposition à la classe qui définit plutôt une description générale de l'objet. Ainsi nous pouvons créer *plusieurs instances* d'un même objet, mais dont les caractéristiques pourront varier légèrement.

Il est important de bien faire la différence entre la *classe*, qui est unique et qui décrit un certain objet, et ses *instances* qui peuvent être multiples et qui sont les objets « réels ». Ceux-ci peuvent être décrits à partir des mêmes propriétés dont les valeurs peuvent varier.

Pour rassurer tout le monde, nous allons prendre un petit exemple concret :

- Une classe `Voiture` permet de définir l'ensemble du parc automobile européen. Cette classe `Voiture` définit l'ensemble des caractéristiques que peut posséder une automobile. Il serait par exemple possible d'y retrouver la `marque` de celle-ci, mais également sa `couleur` ou encore son `immatriculation` que nous pourrions définir comme *attributs*.
- Maintenant nous savons ce qu'est une `Voiture`, nous aurions peut-être l'envie d'en créer quelques-unes. C'est une très bonne idée, et nous allons commencer par créer une `Renault` de `couleur` « grise » et immatriculée dans le « Rhône ». Puis voilà que nous en avons besoin d'une seconde ; très bien, voici une `Peugeot` de `couleur` « noire » et immatriculée dans le « Cantal ».

Dans cet exemple, nous avons donc utilisé une classe `Voiture` pour en créer deux instances qui sont la `Renault` et la `Peugeot`.

Structure

Si vous avez bien suivi le deuxième chapitre de la première partie, vous devriez vous rappeler d'un schéma représentant la structure de la classe principale. Je vous propose de généraliser ce schéma et de le compléter un peu plus avec ce que nous venons de découvrir. Notamment nous pouvons y ajouter les *attributs* et les *méthodes*.
Voici donc comment sont structurées les classes :

Schéma général de la structure d'une classe.

Vous ne devriez donc pas être surpris de la manière dont nous définissons une classe à l'intérieur du code :

Code : Actionscript

```
package
{

    public class NomDeLaClasse
    {
        // Attributs

        // Méthodes

    }

}
```

Nous allons à présent apprendre à définir chacune des propriétés de la classe !

Construire la classe

Package

Avant toute chose, nous allons brièvement reparler de la notion de *package*.
Nous l'avions défini comme décrivant la position de la classe dans l'arborescence des fichiers de votre projet. Rassurez-vous, c'est toujours le cas ! 
Cependant, je voudrais présenter ici la manière à adopter pour positionner ses propres classes dans différents dossiers.

Pour l'instant il est vrai que nous ne disposons pas d'un projet conséquent et que le nombre de classes est assez faible. Cependant lorsque nos projets grandiront et que les classes vont se multiplier, nous devrons faire le tri et ranger nos classes dans différents dossiers afin d'y voir plus clair.

Voici donc un conseil qu'il serait prudent de suivre : *organisez vos classes en fonction de leur nature et de leur utilité !*

Par exemple nous pourrions imaginer un **`package`** nommé `vehicules` qui contiendrait les classes `Voiture`, `Camion` et `Moto`. Puis nous créons deux nouvelles classes `Fourchette` et `Couteau`, qui n'ont apparemment strictement rien à faire dans le **`package`** `vehicules`. Il nous faut alors en insérer un nouveau que nous pourrions nommer `couverts`.

Attributs

Nous allons maintenant présenter le premier type de propriétés d'une classe : il s'agit des *attributs* !
Les attributs ne sont en réalité que des variables, nous pouvons donc les déclarer comme n'importe quelle variable :

Code : Actionscript

```
var _unAttribut:String;
```

Lorsqu'on programme en POO, on a l'habitude de *ne pas initialiser* les variables lors de la déclaration mais plutôt à l'intérieur d'un **constructeur** dont nous discuterons un peu après.

Également nous introduisons le mot-clé **`private`** devant la déclaration de l'attribut, dont nous reparlerons aussi plus tard :

Code : Actionscript

```
private var _unAttribut:String;
```

Ça y est, nous avons notre premier attribut !

Par convention en Actionscript 3, nous ajouterons un caractère *underscore* « _ » devant le nom de tous nos attributs. Cela nous sera pratique lorsque nous écrirons des **accesseurs** et **mutateurs**.

Méthodes

Comme nous l'avons précisé plus haut, les *méthodes* sont des fonctions.
Il peut donc s'agir aussi bien d'*instructions de fonction* que d'*expressions de fonction* (pour ceux qui auraient tout oublié, allez faire discrètement un tour dans le chapitre sur les fonctions ! ). Néanmoins je vous avais dit que les instructions de fonction étaient préférables, nous utiliserons donc ce type de fonctions :

Code : Actionscript

```
function uneMéthode():void
{
    // Instructions
}
```

Bien entendu, à l'intérieur de ces méthodes nous avons accès aux différents attributs afin de pouvoir modifier leur valeur ou simplement lire leur contenu.
Au final, seuls les attributs peuvent *mémoriser* des choses et ainsi se souvenir de l'état de l'objet en question. C'est pourquoi en général vos méthodes serviront à lire ou modifier le contenu d'un ou plusieurs attributs.

Voilà comment notre méthode pourrait modifier un attribut :

Code : Actionscript

```
function uneMéthode(nouvelleValeur:String):void
{
    unAttribut = nouvelleValeur;
}
```

Bien évidemment nous pouvons faire toutes sortes de manipulations à l'intérieur d'une méthode et pas simplement affecter ou lire le contenu d'un attribut.

Contrairement à d'autres langages, l'Actionscript ne prend pas en compte la **surcharge de méthodes**. Pour ceux qui découvrent ce terme, nous en reparlerons avant la fin de ce chapitre.

Enfin, les méthodes prennent également un mot-clé devant leur déclaration, qui est cette fois-ci **`public`** :

Code : Actionscript

```
public function uneMéthode():void
{
    // Instructions
}
```

Les mots-clés **`private`** et **`public`** sont liés à la notion d'**encapsulation** que nous découvrirons plus en profondeur au cours du chapitre. En attendant, sachez que ces mots-clés existent et considérez qu'ils font partie de la déclaration des propriétés d'une classe.

Constructeur

À présent il est temps d'introduire une méthode un peu particulière : le **constructeur** !
Le constructeur est la méthode appelée par défaut lors de l'*initialisation* d'un objet ; vous savez lorsque vous utilisez le mot-clé **`new`**. Vous noterez que cette méthode possède obligatoirement le même nom que celui de la classe.
Par exemple pour notre classe nommée `Voiture`, notre constructeur pourrait ressembler à ceci :

Code : Actionscript

```
public function Voiture()
{
    // Instructions
}
```

Le constructeur d'une classe sert principalement à *initialiser* l'ensemble des attributs déclarés dans celle-ci.

Vous remarquerez que le constructeur ne peut pas renvoyer de valeur, aussi évitez d'insérer le mot-clé **return** à l'intérieur de celui-ci. Une instruction de renvoi serait ignorée à l'exécution, et pourrait entraîner des messages d'erreur lors de la compilation.

Ne vous inquiétez pas si certains points sont encore flous dans votre esprit, nous créerons une classe pas à pas à la fin du chapitre !

Des paramètres facultatifs pour nos méthodes
La surcharge de méthodes

Dans beaucoup de langages utilisant la programmation orientée objet, on retrouve le concept de **surcharge de méthodes**, mais ce n'est pas le cas en ActionScript 3. Contrairement à son nom abominable, ce concept est relativement simple et consiste a définir plusieurs méthodes portant le *même nom*.
Il est alors possible de définir des paramètres de types différents ou encore d'utiliser un nombre de paramètres différent. Ceci est très utile et permet par exemple de définir plusieurs constructeurs n'ayant pas le même nombre de paramètres :

Code : Actionscript

```
package
{

    public class MaClasse
    {
        // Attribut
        private var _unAttribut:int;

        // Constructeurs
        public function MaClasse()
        {
            _unAttribut = 0;
        }

        public function MaClasse(entier:int)
        {
            _unAttribut = entier;
        }

    }

}
```

Dans l'exemple précédent, il serait possible d'instancier une classe sans renseigner de paramètres avec le premier constructeur, ou en précisant la valeur de l'entier avec le second. Cette technique servirait donc à rendre l'utilisation d'une classe plus simple et plus souple, en permettant l'appel « d'une même fonction » avec des paramètres différents.

Malheureusement donc, la surcharge de méthodes est *strictement interdite* en Actionscript, vous ne trouverez donc jamais deux méthodes ayant le même nom au sein d'une classe, et ceci est aussi valable pour les constructeurs. En revanche il est possible de contourner le problème, voire même de le simplifier : la définition de **paramètres facultatifs** !

Les paramètres facultatifs

Définition

Comme nous l'avons déjà dit, la surcharge de méthodes est *interdite* en Actionscript !
C'est un point important, c'est pourquoi ce n'est pas superflu de le redire une nouvelle fois. En revanche en Actionscript il existe un concept qui permet d'obtenir une utilisation similaire. Il s'agit de **paramètres facultatifs** !
Cette nouvelle notion est associée en réalité à la définition de fonctions et n'est donc pas limitée qu'aux méthodes. Ainsi pour insérer un paramètre facultatif, il suffit de lui préciser une valeur par défaut dans lors de la définition de la fonction, comme ceci :

Code : Actionscript

```
function maFonction(entier:int, texte:String = "Valeur par défaut",
nombre:Number = 0):void
{
    trace(nombre, texte);
}
```

En utilisant ici des valeurs par défaut pour les deux derniers paramètres, il alors possible de les « omettre » lors de l'appel de la fonction. Ainsi contrairement à la surcharge de méthode, nous n'avons pas besoin en Actionscript de réécrire l'intégralité du contenu de la fonction pour chaque définition de nouveaux paramètres.

Attention cependant à l'ordre de définition des paramètres ; les paramètres facultatifs doivent obligatoirement être placés à la fin de la liste des paramètres. Également, ils doivent être écrits dans un ordre d'utilisation précis que nous allons préciser en parlant des appels de ces fonctions.

Appels

Comme leur nom l'indique, les paramètres facultatifs peuvent être « omis » lors de l'appel de la fonction. Ainsi la fonction définie précédemment, peut être appelée en renseignant uniquement le premier paramètre :

Code : Actionscript

```
maFonction(10);
```

Il est également possible de renseigner les deux premiers paramètres, sans le troisième :

Code : Actionscript

```
maFonction(10, "Nouveau texte");
```

Attention toutefois à l'ordre de définition des paramètres facultatifs. En effet, s'il est possible de renseigner le premier paramètre facultatif et d'omettre le second, l'inverse n'est pas possible. Ainsi l'appel suivant n'est pas correct : `maFonction(10, 5)`.
C'est pourquoi vous devez être vigilent dans l'ordre de définition des paramètres facultatifs de vos fonctions !

Enfin pour finir, voici dernier appel possible de votre fonction, qui comprend l'intégralité des paramètres :

Code : Actionscript

```
maFonction(10, "Nouveau texte", 5);
```

Ainsi grâce à la définition de paramètres facultatifs, nous avons trois manières différentes d'appeler la même fonction !

Encapsulation

L'**encapsulation** que nous allons redécouvrir maintenant est l'un des concepts les plus importants de la programmation orientée objet !
Vous rappelez-vous du mot-clé **`private`** que nous avons introduit avant chacun de nos attributs ? Il permet de masquer la propriété à laquelle il est associé ; celle-ci n'est donc pas visible depuis l'extérieur de la classe en question. Ainsi à partir de

maintenant, nous masquerons *obligatoirement* l'ensemble de nos attributs !
Pour y accéder, nous serons donc dans l'obligation de faire appel à une méthode intermédiaire qui nous permettra de vérifier les données et d'affecter les attributs en conséquence, hors du champ de vision de l'utilisateur.

L'intérêt de l'*encapsulation* est de simplifier l'utilisation des objets en masquant l'ensemble des attributs et des méthodes qui sont utiles simplement au fonctionnement de l'objet. Ainsi, vu de l'extérieur, nous pourrons manipuler ces objets facilement, *sans nous soucier de leur fonctionnement interne*.

Ce concept introduit donc la notion de **droits d'accès** que nous allons voir tout de suite !

Les différents droits d'accès

Il n'est pas possible de parler d'encapsulation sans toucher un mot des **droits d'accès** ou **portées**. Ces droits d'accès définissent la visibilité d'une propriété au sein de votre code. En Actionscript, il existe trois portées qui sont **`public`**, **`private`** et **`internal`**. Ces mots-clés se placent juste avant la déclaration des propriétés auxquelles ils sont associés.

En réalité, il existe une quatrième portée nommée **`protected`**. Cependant celle-ci est profondément liée à la notion d'**héritage** dont nous reparlons dans un chapitre qui lui est consacré.

Privés

Les droits d'accès dits « privés » s'utilisent avec le mot-clé **`private`**. Ils permettent de restreindre l'utilisation de la propriété à la classe où elle est définie. Ainsi cette propriété ne sera pas visible depuis l'extérieur de cette classe.
Voici un attribut dont la portée est de type **`private`** :

Code : Actionscript

```
private var _monAttribut:String;
```

Je rappelle que *tous nos attributs* doivent être invisibles depuis l'extérieur de la classe où ils sont définis, utilisez donc la portée **`private`**.

Publiques

Les droits d'accès « publiques » sont associés au mot-clé **`public`**, que nous avons déjà croisé plusieurs fois. Celui-ci permet de rendre visible partout dans le code la propriété à laquelle il est associé. Ce sera donc le cas pour la majorité de vos méthodes.
D'ailleurs, il est *impossible* d'affecter une autre portée que **`public`** à un constructeur :

Code : Actionscript

```
public function MonConstructeur():void
{
    ...
}
```

Internes

Les droits d'accès « internes » sont un peu spéciaux ; ils sont associés au mot-clé **`internal`**. Les propriétés définies avec ce type de portées sont visibles depuis l'ensemble du package, dont la classe où elle est déclarée appartient.

Code : Actionscript

```
internal var _monAttribut:int;
```

Cette portée n'est pas très utilisée en général, mais sachez qu'il s'agit de la portée par défaut lorsqu'aucune autre n'est spécifiée.

Les accesseurs

Syntaxe

Il existe un type de méthodes un peu spécial, qui est directement lié à la notion d'encapsulation : les **accesseurs** !
En réalité, on désigne par accesseurs l'ensemble des *accesseurs* et *mutateurs*, également appelées **getters** et **setters**. Ceux-ci permettent l'accès direct à un attribut de portée **private** en lecture par l'accesseur et en écriture par le mutateur. Ainsi dans beaucoup de langages de programmation, on retrouve un ensemble d'accesseurs dont le nom débute par `get` (de l'anglais *to get* qui signifie « obtenir ») et un ensemble de mutateurs dont le nom débute par `set` (de l'anglais *to set* qui signifie « définir »).

Pour gérer l'ensemble des accesseurs, l'Actionscript a introduit deux mots-clés **get** et **set** qui permettent notamment de donner un nom identique aux deux accesseurs.
Pour illustrer ça d'un exemple, voici un attribut quelconque dont la portée est de type **private** :

Code : Actionscript

```
private var _texte:String;
```

Étant donné que cet attribut est de type **private**, il n'est pas accessible depuis l'extérieur de la classe où il a été défini. Cependant, il est probable que nous ayons besoin de modifier cet attribut depuis la classe principale. Si nous voulons respecter le concept d'encapsulation, nous devons donc conserver la portée de cet attribut et définir des accesseurs pour y avoir accès :

Code : Actionscript

```
// Accesseur
public function get texte():String
{
    return _texte;
}

// Mutateur
public function set texte(nouveauTexte:String):void
{
    _texte = nouveauTexte;
}
```

Comme vous le voyez, mes deux accesseurs utilisent le même nom de fonction, ce qui est drôlement pratique. Mais l'utilisation des accesseurs va plus loin que ça, car leur utilisation est un peu particulière. Ainsi l'accesseur **get** d'une instance nommée `MonObjet` s'utilise sans les parenthèses :

Code : Actionscript

```
var maVariable:String = MonObjet.texte;
```

Quant à lui, le mutateur s'utilise également sans parenthèses, mais avec le symbole égal « = » qui est considéré comme un signe d'affectation :

Code : Actionscript

```
MonObjet.texte = "Nouvelle chaîne de caractères";
```

Une telle utilisation des accesseurs est spécifique à l'Actionscript. Cette technique permet de manipuler les accesseurs comme s'il s'agissait de l'attribut lui-même. Cependant en utilisant les *getters* et *setters*, vous respectez le concept

d'*encapsulation*. Ainsi, les propriétés créées à partir d'accesseurs sont considérées comme étant des attributs, et non des méthodes.

Une raison supplémentaire d'utiliser les accesseurs

Lorsque vous utilisez des accesseurs, vous n'êtes pas obligés de vous contenter de lire un attribut dans un *getter* ou de lui affecter une nouvelle valeur dans un *setter* : en effet, il est tout à fait possible d'ajouter du code supplémentaire, voire de ne pas manipuler d'attribut en particulier !

Prenons la classe suivante :

Code : Actionscript

```
package
{
    /**
* Une classe d'exemple
*/
    public class Voiture
    {
        private var _largeur:int;

        public function Voiture()
        {

        }

        public function get largeur():int
        {
            return _largeur;
        }

        public function set largeur(value:int):void
        {
            _largeur = value;
        }

        public function mettreAJour():void
        {
            // Mettre à jour l'affichage
        }
    }

}
```

Dans un premier temps, imaginons que dans notre classe, nous disposions d'une méthode `mettreAJour()` qui met à jour l'affichage de notre objet en fonction de la valeur de l'attribut `_largeur`. Pour spécifier la largeur de la voiture, nous procéderions ainsi :

Code : Actionscript

```
var voiture:Voiture = new Voiture();
voiture.largeur = 100; // Largeur de la voiture
voiture.mettreAJour(); // Mettons à jour l'affichage de la voiture
pour une largeur de 100
voiture.largeur = 150; // Changeons la largeur de la voiture
voiture.mettreAJour(); // Mettons à jour l'affichage de la voiture
pour une largeur de 150
```

Grâce aux accesseurs, il est possible de l'appeler automatiquement dès que l'on modifie la largeur de l'objet :

Code : Actionscript

```
public function set largeur(value:int):void
{
    _largeur = value;
    mettreAJour();
}
```

Ainsi, au lieu d'avoir à appeler manuellement la méthode `mettreAJour()`, il suffit de modifier la largeur :

Code : Actionscript

```
var voiture:Voiture = new Voiture();
voiture.largeur = 100; // Largeur de la voiture
// L'affichage de la voiture est automatiquement mis à jour dans
l'accesseur set largeur !
voiture.largeur = 150; // Changeons la largeur de la voiture
// Encore une fois, l'affichage de la voiture est automatiquement
mis à jour, il n'y a rien d'autre à faire !
```

Maintenant, nous aimerions limiter les valeurs possibles de l'attribut largeur ; disons qu'il doit être supérieur à 100 et inférieur à 200.

Code : Actionscript

```
var voiture:Voiture = new Voiture();
voiture.largeur = 100; // Largeur de la voiture
// On vérifie que la largeur est dans les limites
if(voiture.largeur < 100)
{
    voiture.largeur = 100;
}
else if(voiture.largeur > 200)
{
    voiture.largeur = 200;
}
trace(voiture.largeur); // Affiche: 100

voiture.largeur = 250; // Changeons la largeur de la voiture
// On vérifie une fois de plus que la largeur est dans les limites
if(voiture.largeur < 100)
{
    voiture.largeur = 100;
}
else if(voiture.largeur > 200)
{
    voiture.largeur = 200;
}
trace(voiture.largeur); // Affiche: 200
```

Vous remarquerez que c'est plutôt fastidieux. Bien sûr, nous pourrions utiliser une fonction, mais il vaut mieux mettre dans la classe `Voiture` ce qui appartient à la classe `Voiture` ! 
Encore une fois, les accesseurs nous facilitent grandement la tâche ; voyez plutôt :

Code : Actionscript

```
public function set largeur(value:int):void
{
    _largeur = value;

    // _largeur doit être comprise entre 100 et 200
    if (_largeur < 100)
    {
        _largeur = 100;
```

```
        }
        else if(_largeur > 200)
        {
            _largeur = 200;
        }

        mettreAJour();
    }
```

Le code principal serait alors écrit ainsi :

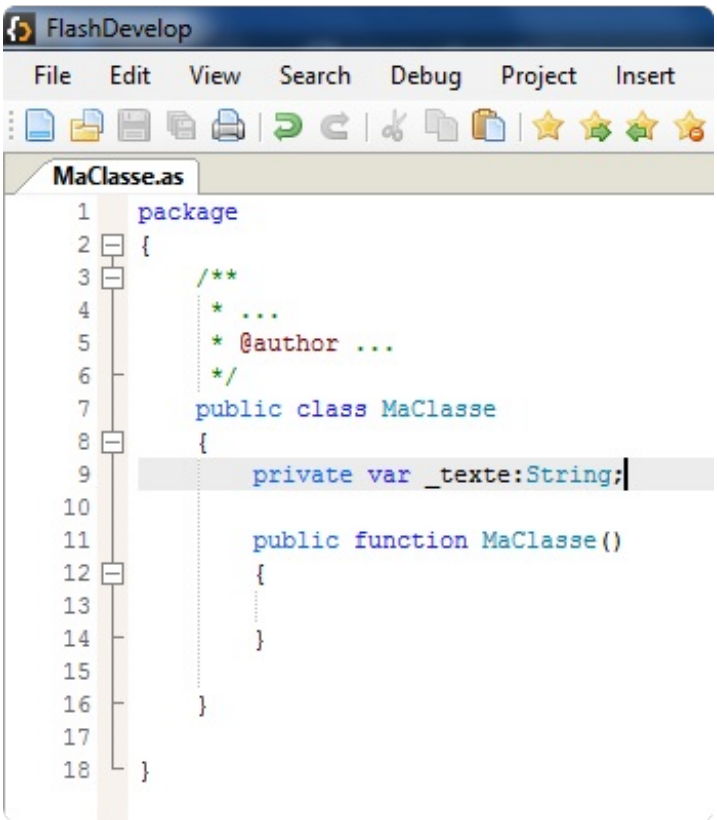
Code : Actionscript

```
var voiture:Voiture = new Voiture();
voiture.largeur = 100; // Largeur de la voiture
// Plus besoin de vérifier que la largeur est dans les limites,
l'accesseur le fait pour nous !
trace(voiture.largeur); // Affiche: 100

voiture.largeur = 250; // Changeons la largeur de la voiture
trace(voiture.largeur); // Affiche: 200
```

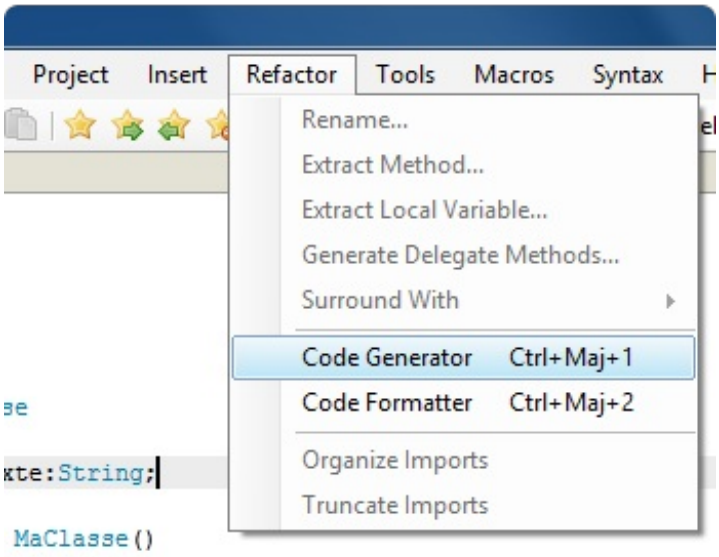
Avouez que c'est extrêmement pratique ! Je vous conseille d'appliquer cette façon de faire le plus souvent possible, cela vous rendra service.

Générateur de code dans Flashdevelop

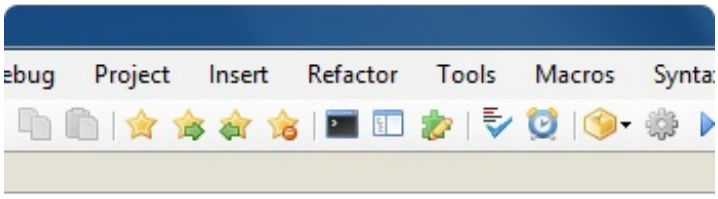
Il existe un outil très pratique disponible dans la plupart des IDE, dont Flashdevelop, pour générer automatiquement des portions de codes. Nous pouvons l'utiliser, entre autres, pour générer les accesseurs de nos attributs ! Pour cela, il faut suivre trois étapes :

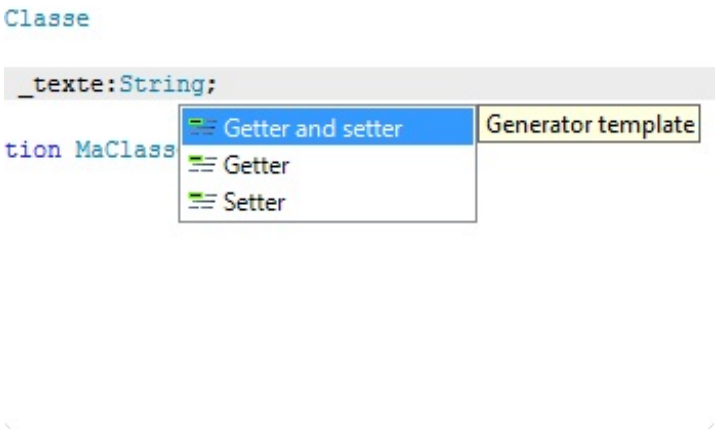


1. Placer le curseur sur la ligne de l'attribut dont il faut générer les accesseurs



2. Sélectionner l'option 'Code Generator'





3. Sélectionner une des trois options

Il est également possible de se passer de la deuxième étape en utilisant directement le raccourci Ctrl + Maj + 1.

Le code ainsi généré, ressemblera à ceci :

Code : Actionscript

```
public function get texte():String
```

```
        {
            return _texte;
        }

        public function set texte(value:String):void
        {
            _texte = value;
        }
```

Exercice : Créons notre première classe

Présentation de la classe

Description

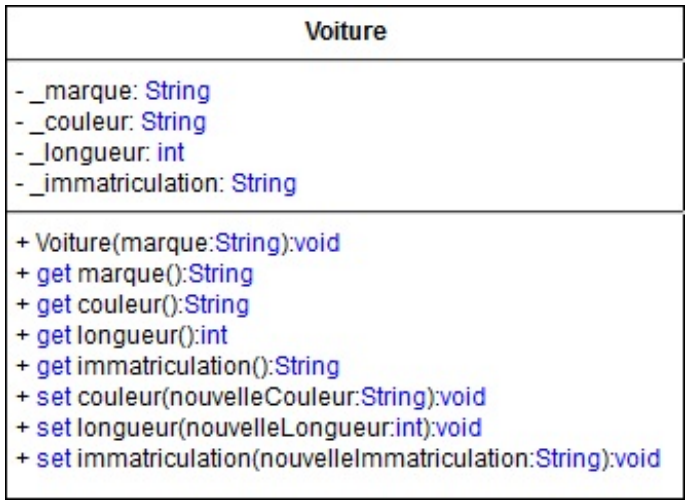
Afin de mieux comprendre tout ceci, nous allons maintenant écrire notre première classe pas à pas ! Nous allons donc créer une classe `Voiture`, que je vous propose de découvrir maintenant. Nous y intégrerons les attributs suivants, ainsi que les accesseurs correspondants :

- `saMarque` : cet argument de type `String` permet de définir la marque de la voiture. Celui-ci sera défini dans le constructeur et il s'agit du seul attribut qui ne possèdera pas de mutateur. En effet, une voiture peut être repeinte, modifiée par divers « accessoires » qui influeront sur sa longueur ou encore elle peut changer de plaque d'immatriculation. En revanche, elle ne peut pas changer sa marque de fabrique.
- `saCouleur` : cet argument de type `String` représente la couleur de peinture de la voiture. Celle-ci sera manipulée par les deux accesseurs nommés `couleur`.
- `saLongueur` : la longueur du véhicule sera définie par une variable de type `int`. Deux accesseurs `longueur` permettront de manipuler cet élément. Lors d'une affectation, nous devrons vérifier si la valeur renseignée est positive. Dans le cas contraire, nous utiliserons la valeur `-1` pour préciser que la longueur est « non renseignée ».
- `sonImmatriculation` : enfin l'immatriculation du véhicule sera stockée dans un attribut de type `String`, qui possèdera deux accesseurs `immatriculation`.

Lançons-nous donc dans la conception de cette classe. Lisez avec attention, afin de noter les différentes étapes de création d'une classe.

UML : Unified Modeling Language

Lorsqu'on programme, il est possible de représenter les différentes classes sur un schéma pour « résumer » leurs propriétés.
Il existe différentes modélisations standards dont l'**Unified Modeling Language** ou **UML**. Cette représentation est très souvent associée aux langages orientés objets comme l'Actionscript.
Sans donner plus de détails, je vous propose de découvrir la représentation correspondant à notre classe `Voiture` :



Représentation UML de la classe Voiture

Comme vous le voyez, notre classe est divisée en deux parties : les attributs et les méthodes !
Ceux-ci sont donc listés en spécifiant de quels types ils sont, ainsi que les paramètres à renseigner en ce qui concerne les méthodes. Cela permet de dresser une sorte de « plan de construction » de la classe à coder, mais également son mode d'emploi pour d'éventuels autres programmeurs. Cela permet également de mettre en évidence les relations liant les classes les unes aux autres, comme nous le verrons au cours des chapitres à venir.
Vous aurez très certainement remarqué les signes « - » et « + » qui précèdent l'ensemble de nos propriétés. Ceux-ci permettent de représenter les différents droits d'accès liés à chacune de nos propriétés de la manière suivante : « - » pour privés, « + » pour

publiques, « ~ » pour internes et « # » pour protégés.

L'objectif du cours n'est pas de vous apprendre à utiliser l'UML, cependant nous utiliserons quelques schémas qui sont souvent plus clairs que des mots. Vous apprendrez à lire et utiliser les bases de cette modélisation au fil des chapitres de cette partie.

Écriture du code

Préparation du nouveau fichier

Tout d'abord, pour créer une nouvelle classe, nous aurons besoin d'un *nouveau fichier Actionscript* !
Nous allons donc insérer une classe nommée `Voiture` avec `File > New > AS3 Document` si vous êtes sous *FlashDevelop* ou créer un nouveau fichier nommé `Voiture.as` si vous n'utilisez pas ce logiciel.

Puis, nous insèrerons à l'intérieur le code de définition de la classe, comme ceci :

Code : Actionscript

```
package
{
    public class Voiture
    {

    }

}
```

Déclaration des attributs

Précédemment, nous avons défini les quatre attributs qui sont `saMarque`, `saCouleur`, `saLongueur` et `sonImmatriculation`. Tous ces attributs ont évidemment des droits d'accès de type **`private`**, pour respecter le concept d'encapsulation.

Voici donc les différentes déclarations d'attributs :

Code : Actionscript

```
private var _marque:String;
private var _couleur:String;
private var _longueur:int;
private var _immatriculation:String;
```

Le constructeur

Comme nous l'avons rapidement introduit, le constructeur de cette classe devra recevoir en paramètre la marque de fabrication du véhicule. Sachant que les autres attributs possèdent des *setters*, nous ne les introduirons pas dans la liste des paramètres à spécifier au constructeur. Néanmoins, ceux-ci devront tout de même être initialisés.

Il est recommandé d'utiliser dès que cela est possible les accesseurs et les mutateurs au sein même de la classe ; exception faite du constructeur, où cela n'est généralement pas nécessaire.

Découvrons tout de suite ce constructeur :

Code : Actionscript

```
public function Voiture(marque:String)
```

```
{
    _marque = marque;
    _couleur = "Sans couleur";
    _longueur = -1;
    _immatriculation = "Non immatriculée";
}
```

Vous pouvez utiliser des noms de paramètres identiques aux noms de vos propriétés (ici `marque`) : ils sont en effet prioritaires. Pour pouvoir utiliser explicitement une propriété de l'objet, il faut ajouter le mot-clé **`this`** devant : si nous avions un mutateur pour l'attribut `marque`, nous devrions écrire **`this`**`.marque = marque;` pour l'utiliser.

Les accesseurs

Chaque attribut possède un ou deux accesseurs, nous ne les détaillerons donc pas tous. Je vous propose plutôt de découvrir un *getter* et un *setter*.
Nous allons prendre l'exemple des accesseurs `longueur`, dont voici le *getter* :

Code : Actionscript

```
public function get longueur():int
{
    return _longueur;
}
```

Cet accesseur n'a rien de très compliqué, nous n'en parlerons donc pas plus. En revanche pour le mutateur, nous devons vérifier si la valeur spécifiée est positive, je vous rappelle. Nous devrons donc utiliser une condition en **`if...else`** pour faire la vérification.

Voici donc le mutateur en question, que vous êtes normalement en mesure de comprendre par vous-mêmes maintenant :

Code : Actionscript

```
public function set longueur(nouvelleLongueur:int):void
{
    if(nouvelleLongueur > 0)
        _longueur =  nouvelleLongueur;
    else
        _longueur =  -1;
}
```

Félicitations, nous avons terminé l'écriture de votre première classe !

La classe complète

Parce qu'il est probable que certains ne soient pas pleinement satisfaits avant d'avoir vu l'intégralité de celle-ci, je vous propose ici un récapitulatif intégral de la classe `Voiture` :

Code : Actionscript - Voiture

```
package
{
    public class Voiture
    {
        /*************** Attributs ***************/
        private var _marque:String;
        private var _couleur:String;
        private var _longueur:int;
        private var  immatriculation:String;
```

```
            /************* Constructeur *************/
            public function Voiture(marque:String)
            {
                _marque = marque;
                _couleur = "Sans couleur";
                _longueur = -1;
                _immatriculation = "Non immatriculée";
            }

            /*************** Accesseurs ***************/
            public function get marque():String
            {
                return _marque;
            }

            public function get couleur():String
            {
                return _couleur;
            }

            public function get longueur():int
            {
                return _longueur;
            }

            public function get immatriculation():String
            {
                return _immatriculation;
            }

            /*************** Mutateurs ***************/
            public function set couleur(nouvelleCouleur:String):void
            {
                _couleur = nouvelleCouleur;
            }

            public function set longueur(nouvelleLongueur:int):void
            {
                _longueur = (nouvelleLongueur > 0) ? nouvelleLongueur :
-1;
            }

            public function set
immatriculation(nouvelleImmatriculation:String):void
            {
                _immatriculation = nouvelleImmatriculation;
            }

        }

}
```

En résumé

- Le mot-clé **`this`** fait référence à l'objet lui-même.
- Les classes sont triées dans différents **`package`** en fonction du type d'objets qu'elles représentent.
- Le **constructeur** est une méthode particulière appelée à la création d'une instance, où on initialise généralement les attributs.
- Il est possible de définir des **paramètres facultatifs** à l'intérieur de nos fonctions et méthodes, pour palier l'impossibilité de **surcharger** celles-ci.
- L'**encapsulation** est le concept permettant de masquer le fonctionnement interne d'une classe.
- Pour gérer les **droits d'accès**, nous disposons des différents mots-clés **`public`**, **`private`**, **`internal`** et **`protected`** (que nous aborderons plus loin).
- Des **accesseurs** peuvent être déclarés à l'aide des mots-clés **`get`** et **`set`**, et simplifient l'accès aux attributs d'une classe.

Les classes (2nde partie)

Dans le chapitre précédent, nous avons présenté les bases de la théorie des *classes*.
Nous allons maintenant introduire des notions complémentaires, qui vous permettront de structurer votre code encore plus facilement et proprement. Contrairement à la plupart des notions de cette partie sur la POO, celles que nous allons découvrir dans ce chapitre n'ont pas été présentées dans le premier chapitre, et sont donc *complètement nouvelles*.
Redoublez donc d'attention !

Les éléments statiques

Peut-être que certains se rappellent de la classe `Math` que nous avons déjà utilisé. Ils se demandent alors pourquoi nous l'avions utilisé en utilisant directement le nom de la classe et *sans passer par une instance* de celle-ci :

Code : Actionscript

```
var monNombre:Number = Math.sqrt(2);
```

Comment cela est-ce possible ?

Maintenant que vous avez déjà quelques connaissances en POO, il est tout à fait justifié de se poser ce genre de question. En fait, la classe `Math` utilise des éléments qui sont un peu particuliers : les **éléments statiques** !
Comme nous le verrons, ces éléments statiques ne sont pas définis pour les instances d'une classe, mais pour la classe elle-même. Il existe deux types d'éléments statiques qui sont :

- les variables statiques
- les méthodes statiques.

Nous verrons donc comment créer ces éléments et quel est leur intérêt.

Les variables statiques

Les variables statiques sont déclarées à l'aide du mot-clé **`static`**, et sont associées donc définies pour la classe.
Prenons l'exemple de notre classe `Voiture` du chapitre précédent, et ajoutons-y une variable statique représentant le nombre de fois où celle-ci a été instanciée :

Code : Actionscript

```
public static var occurences:int = 0;
```

Cette variable est donc *partagée* par la classe, elle n'appartient pas aux *instances* de celle-ci. Toutefois cette variable est accessible depuis n'importe quel point de la classe. Notamment, nous pourrions incrémenter cette variable à l'intérieur du constructeur de la classe `Voiture` afin de comptabiliser le nombre d'occurrences de celle-ci :

Code : Actionscript

```
occurrences++;
```

Grâce à cette variable statique, nous pourrions obtenir le nombre d'instances de la classe `Voiture`, n'importe où dans le code.
Pour cela, nullement besoin de créer une nouvelle instance de la classe, il suffit d'utiliser le nom de la classe lui-même :

Code : Actionscript

```
var uneRenault:Voiture = new Voiture("Renault");
var unePeugeot:Voiture = new Voiture("Peugeot");
var uneCitroen:Voiture = new Voiture("Citroën");
trace(Voiture.occurrences); // Affiche : 3
```

Un élément statique ne peut être utilisé qu'avec la classe où celui-ci est déclaré. Il est impossible de faire référence à un élément statique à l'aide d'une instance de cette classe : des erreurs seraient alors engendrées.

Les méthodes statiques

Il existe un second type d'éléments statiques : il s'agit des *méthodes statiques*.
Dans le chapitre précédent, je vous avais dit que les méthodes servaient principalement à la lecture ou à la modification d'un attribut. Nous pouvons donc introduire les méthodes statiques comme l'ensemble des méthodes qui offrent des fonctionnalités n'affectant pas au moins l'un des attributs d'une classe.

Ces éléments statiques sont également déclarés à l'aide du mot-clé **`static`** :

Code : Actionscript

```
public static function uneMethode():void
{
    // Instructions
}
```

À l'aide de ces méthodes statiques, il nous est possible de recréer la classe `Math`, que nous pourrions renommer `MaClasseMath`. Voici par exemple la redéfinition de la méthode `pow()` en `puissance()` :

Code : Actionscript

```
public static function puissance(nombre:int, exposant:int):int
{
    var resultat:int = nombre
    for(var i:int = 1; i < exposant; i++)
    {
        resultat *= nombre;
    }
    return resultat;
}
```

Le code complet de la classe serait :

Code : Actionscript

```
package
{
    public class MaClasseMath
    {
        public static function puissance(nombre:int,
exposant:int):int
        {
            var resultat:int = nombre
            for(var i:int = 1; i < exposant; i++)
            {
                resultat *= nombre;
            }
            return resultat;
        }
    }
}
```

Vous remarquerez que cette classe ne possède pas de constructeur : en effet, il est permis de ne pas mettre de constructeur si vous ne vous en servirez pas. Ici, nous n'avons pas mis de constructeur car nous n'allons jamais créer d'instance de cette classe.

Nous pouvons ainsi l'utiliser sans créer d'occurrences de cette nouvelle classe :

Code : Actionscript

```
var monNombre:int = MaClasseMath.puissance(2,10);
trace(monNombre); // Affiche : 1024
```

Des classes telles que `Math` ont été conçues pour être utilisées uniquement grâce à des éléments statiques. En utilisant ce principe, vous pouvez ainsi regrouper un ensemble de fonctionnalités à l'intérieur d'une même classe. Vous n'aurez donc pas besoin de créer d'occurrences de celle-ci et vous ferez ainsi l'économie des instructions de déclarations et d'initialisations des instances.

Il est impossible d'utiliser le mot-clé **this** dans les méthodes statiques, car elles ne sont liées à aucun objet en particulier.

Une nouvelle sorte de « variable » : la constante !

Lorsque nous avons introduit les variables dans la première partie, nous n'avons pas parlé des **constantes** !
Comme son nom l'indique, la valeur d'une constante est *figée* contrairement à celle d'une variable qui est vouée à évoluer au cours du programme. Ces constantes sont principalement utilisées en POO, et représentent des caractéristiques constantes d'un objet.

Je vous invite à découvrir ce nouveau type d'élément sans plus attendre !

Présentation

Déclaration

De la même façon que nous avions l'instruction ou mot-clé **var** pour déclarer une variable, nous disposons du mot-clé **const** en ce qui concerne les constantes. Ces dernières possèdent également un type, de la même manière que les variables.
Voici par exemple la déclaration d'une constante de type `String` :

Code : Actionscript

```
const MA_CONSTANTE:String;
```

Vous remarquerez qu'ici nous n'utilisons pas la notion Camel. Effectivement, il est de coutume d'écrire les noms de constantes en lettres majuscules. Cela permet de les différentier des variables, et de préciser qu'il s'agit bien d'une constante.
Utilisez l'underscore « _ » pour séparer les différents mots à l'intérieur du nom de votre constante.

Le code précédent n'a malheureusement aucun intérêt et ne sert à rien sans l'*initialisation* de la constante !

Initialisation

Tout comme une variable, il serait bien d'initialiser une constante. Vous pouvez procéder exactement de la même manière que pour une variable. La technique d'initialisation dépend bien entendu du type de la constante.

Voici donc comment initialiser notre constante précédente de type `String` :

Code : Actionscript

```
const MA_CONSTANTE:String = "Valeur";
```

Contrairement aux variables, Il est strictement *impossible* d'initialiser la valeur d'une constante ailleurs que lors de sa déclaration. Étant donné que la valeur d'une constante est non modifiable, n'essayez pas non plus de procéder à une affectation.

Intérêt des constantes

Il y a certainement plus de la moitié, voire même les trois quarts d'entre vous qui se sont posé la question suivante :

À quoi ces constantes peuvent-elles bien servir ?

Contrairement à ce que vous pensez, les constantes ont plusieurs utilités.

- Tout d'abord, elles permettent de *mettre des noms* sur des valeurs. Votre programme ne marchera pas mieux avec cela, c'est uniquement une question de clarification du code. Avouez qu'il est quand même plus aisé de comprendre la signification d'une expression, si celle-ci utilise des noms plutôt que des valeurs : `prixRoue * NOMBRE_DE_ROUES` plutôt que `prixRoue * 4`.
  Dans le second cas, nous pourrions nous demander s'il s'agit d'une augmentation du prix d'une roue, une conversion du prix des euros aux dollars, ou bien une multiplication par le nombre de roues. Dans la première expression, l'opération est tout à fait claire ; ce qui simplifie grandement le travail de relecture d'un code.
- Une autre utilité des constantes est de s'assurer de la pérennisation du code. Imaginez que le nombre de roues de votre voiture puisse servir à plusieurs calculs comme le prix de l'ensemble, son poids, ... Vous devrez donc utiliser cette valeur plusieurs fois dans votre code et à des endroits différents. Ainsi en utilisant une constante à la place de la valeur réelle, vous facilitez une éventuelle mise à jour de votre programme dans l'hypothèse de l'invention de la voiture à 6 roues ! Essayez d'imaginer le travail qu'il aurait fallu fournir pour remplacer chacune des valeurs présentes dans des coins opposés de votre code.

N'utilisez pas non plus des constantes à tour de bras dans vos programmes. Leur but est de simplifier la lecture du code ; n'allez donc pas le compliquer davantage en remplaçant n'importe quelle valeur par une constante !

Un objet dans un objet (dans un objet...)

Jusqu'à présent, nous n'avions utilisé qu'une seule classe à la fois. Mais là où la POO devient vraiment intéressante, c'est lorsque nous combinons les classes entre elles !

Le problème du pétrole

Reprenons la classe `Voiture` :

Code : Actionscript

```
package
{
    public class Voiture
    {
        /*************** Attributs ***************/
        private var _marque:String;
        private var _couleur:String;
        private var _longueur:int;
        private var _immatriculation:String;

        /************* Constructeur *************/
        public function Voiture(marque:String)
        {
            _marque = marque;
            _couleur = "Sans couleur";
            _longueur = -1;
            _immatriculation = "Non immatriculée";
        }

        /*************** Accesseurs ***************/
        public function get marque():String
        {
            return _marque;
        }

        public function get couleur():String
        {
            return _couleur;
```

```
            }

            public function get longueur():int
            {
                return _longueur;
            }

            public function get immatriculation():String
            {
                return _immatriculation;
            }

            /*************** Mutateurs ***************/
            public function set couleur(nouvelleCouleur:String):void
            {
                _couleur = nouvelleCouleur;
            }

            public function set longueur(nouvelleLongueur:int):void
            {
                _longueur = (nouvelleLongueur > 0) ? nouvelleLongueur :
-1;
            }

            public function set
immatriculation(nouvelleImmatriculation:String):void
            {
                _immatriculation = nouvelleImmatriculation;
            }

        }

    }
```

Nous voulons à présent que nos objets contiennent de l'essence. Pour cela, nous serions tenté de procéder ainsi :

Code : Actionscript

```
    package
    {
        public class Voiture
        {
            /*************** Attributs ***************/
            private var _marque:String;
            private var _couleur:String;
            private var _longueur:int;
            private var _immatriculation:String;

            private var _typeEssence:String;
            private var _prixEssence:Number;
            private var _quantiteEssence:Number;

            /************* Constructeur *************/
            public function Voiture(marque:String)
            {
                _marque = marque;
                _couleur = "Sans couleur";
                _longueur = -1;
                _immatriculation = "Non immatriculée";
                _typeEssence = "Sans Plomb";
                _prixEssence = 1.4; // Euros par litre
                _quantiteEssence = 10; // Litres
            }

            /*************** Accesseurs ***************/
            public function get marque():String
            {
```

```
                return _marque;
            }

            public function get couleur():String
            {
                return _couleur;
            }

            public function get longueur():int
            {
                return _longueur;
            }

            public function get immatriculation():String
            {
                return _immatriculation;
            }

            public function get typeEssence():String
            {
                return _typeEssence;
            }

            public function get prixEssence():Number
            {
                return _prixEssence;
            }

            public function get quantiteEssence():Number
            {
                return _quantiteEssence;
            }

            /*************** Mutateurs ***************/
            public function set couleur(nouvelleCouleur:String):void
            {
                _couleur = nouvelleCouleur;
            }

            public function set longueur(nouvelleLongueur:int):void
            {
                _longueur = (nouvelleLongueur > 0) ? nouvelleLongueur :
-1;
            }

            public function set
immatriculation(nouvelleImmatriculation:String):void
            {
                _immatriculation = nouvelleImmatriculation;
            }

            public function set typeEssence(nouveauType:String):void
            {
                _typeEssence = nouveauType;
            }

            public function set prixEssence(nouveauPrix:Number):void
            {
                _prixEssence = nouveauPrix;
            }

            public function set
quantiteEssence(nouvelleQuantite:Number):void
            {
                _quantiteEssence = nouvelleQuantite;
            }

        }

    }
```

Notre classe commence à devenir compliquée ; il vaudrait mieux créer une nouvelle classe pour partager les propriétés.

Une nouvelle classe

Créons une classe `Essence` à mettre dans le fichier `Essence.as` :

Code : Actionscript

```
package
{
    public class Essence
    {
        /*************** Attributs ***************/
        private var _type:String;
        private var _prix:Number;
        private var _quantite:Number;

        /************* Constructeur *************/
        public function Essence()
        {
            _type = "Sans Plomb";
            _prix = 1.4; // Euros par litre
            _quantite = 10; // Litres
        }

        /*************** Accesseurs ***************/
        public function get type():String
        {
            return _type;
        }

        public function get prix():Number
        {
            return _prix;
        }

        public function get quantite():Number
        {
            return _quantite;
        }

        /*************** Mutateurs ***************/
        public function set type(nouveauType:String):void
        {
            _type = nouveauType;
        }

        public function set prix(nouveauPrix:Number):void
        {
            _prix = nouveauPrix;
        }

        public function set quantite(nouvelleQuantite:Number):void
        {
            _quantite = nouvelleQuantite;
        }

    }

}
```

Nous transférons donc toutes les propriétés relatives à l'essence de la voiture dans la nouvelle classe. Il va falloir maintenant adapter la classe `Voiture` :

Code : Actionscript

```
package
{
    public class Voiture
    {
        /*************** Attributs ***************/
        private var _marque:String;
        private var _couleur:String;
        private var _longueur:int;
        private var _immatriculation:String;

        private var _carburant:Essence; // Nouvel attribut pointant
sur un objet de classe Essence !

        /************* Constructeur *************/
        public function Voiture(marque:String)
        {
            _marque = marque;
            _couleur = "Sans couleur";
            _longueur = -1;
            _immatriculation = "Non immatriculée";
            _carburant = new Essence(); // Nous créons un objet
Essence par défaut dans le constructeur
        }

        /*************** Accesseurs ***************/
        public function get marque():String
        {
            return _marque;
        }

        public function get couleur():String
        {
            return _couleur;
        }

        public function get longueur():int
        {
            return _longueur;
        }

        public function get immatriculation():String
        {
            return _immatriculation;
        }

        public function get carburant():Essence // Nouvel accesseur,
renvoyant un objet de classe Essence
        {
            return _carburant;
        }

        /*************** Mutateurs ***************/
        public function set couleur(nouvelleCouleur:String):void
        {
            _couleur = nouvelleCouleur;
        }

        public function set longueur(nouvelleLongueur:int):void
        {
            _longueur = (nouvelleLongueur > 0) ? nouvelleLongueur :
-1;
        }

        public function set
immatriculation(nouvelleImmatriculation:String):void
        {
            _immatriculation = nouvelleImmatriculation;
```

```
            }

            public function set carburant(nouveauCarburant:Essence):void
    // Nouveau mutateur, affectant un objet de classe Essence
            {
                _carburant = nouveauCarburant;
            }
        }

    }
```

Comme vous pouvez le constater, nous pouvons écrire des attributs pointant sur des objets. Nous pourrions même mettre un attribut de type `Voiture` !

Code : Actionscript

```
private var ancienneVoiture:Voiture; // L'ancienne voiture du
propriétaire
```

Pour modifier le carburant de notre voiture, il faut procéder ainsi :

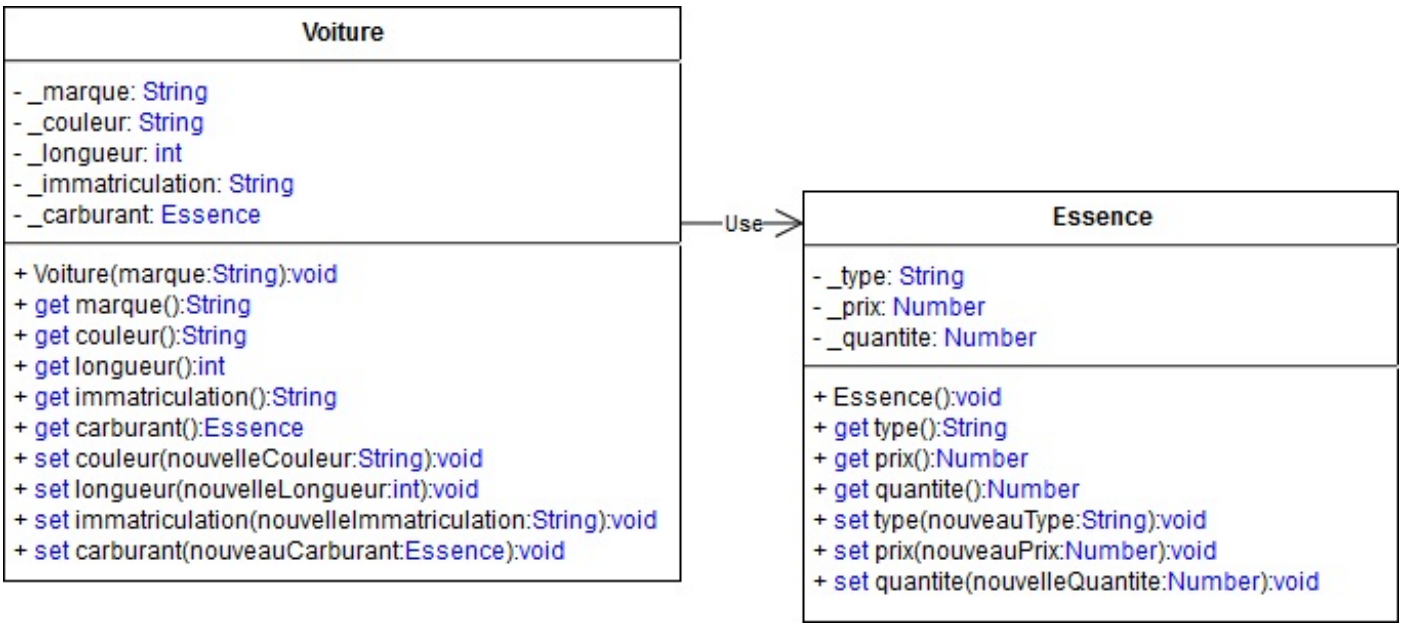
Code : Actionscript

```
var maVoiture = new Voiture("Peugeot");
maVoiture.carburant.type = "Diesel";
trace("Type de carburant : " + maVoiture.carburant.type); // Affiche
: Type de carburant : Diesel
```

Vous remarquerez que nous procédons de la même façon que pour toutes les propriétés, en utilisant le caractère *point* « . », comme nous l'avons vu dans le premier chapitre cette partie. Il suffit donc de mettre un point à chaque fois que nous voulons accéder à la propriété d'un objet :

- une première fois lorsque nous voulons accéder à la propriété `carburant` de notre objet `maVoiture`,
- une seconde fois lorsque nous voulons modifier le type du carburant de la voiture.

Pour résumer la situation, je vous propose un petit schéma UML des classes `Voiture` et `Essence` que nous venons de créer :



Les classes Voiture et Essence

En réalité, nous combinons depuis le début la classe `Voiture` et la classe `String` : beaucoup de nos attributs sont du type `String`. Rappelez-vous : les chaînes de caractères sont aussi des objets !

Exercice : Jeu de rôle

Présentation de l'exercice

Le combat final contre le grand Méchant approche ! Votre personnage, son épée légendaire au poing, se dresse devant cet immense monstre armé jusqu'aux dents ! 

Le moment est venu de faire un peu de pratique ! 
Je propose la réalisation d'un petit programme ressemblant à un jeu de rôle, afin de bien revoir les notions du chapitre.

L'objectif de cet exercice est de créer la ou les classes nécessaires au bon fonctionnement du programme principal (que nous adapterons si besoin). Voici le déroulement de ce programme :

- Nous créons un objet représentant votre personnage, puis nous l'armons.
- Nous créons de façon similaire l'objet représentant le Méchant.
- Le Méchant attaque une fois votre personnage.
- Votre personnage riposte et attaque une fois le Méchant.

Pour apporter un peu de piment à ce programme, les personnages peuvent succomber aux attaques, et leur arme peut infliger un *coup critique* (elle a des chances d'infliger le double de dégâts à l'adversaire).

Nous allons passer par trois étapes que j'ai nommée *solutions*, pour voir progressivement comment programmer correctement en Orienté-Objet : à chaque étape, nous améliorerons notre code.


Il ne s'agit pas d'un TP : nous allons programmer ensemble et progressivement.

Solution initiale

Création de la classe

Commençons par créer notre première classe : j'ai choisi de l'appeler `Personnage`. En effet, ce sera la classe des objets représentant nos deux personnages (vous et le Méchant).
Dans un nouveau fichier appelé `Personnage.as`, écrivons la structure de base de notre classe : le *package*, la classe et le constructeur :

Code : Actionscript

```
package
{

    public class Personnage
    {

        // Constructeur
        public function Personnage()
        {

        }
    }
}
```

Les attributs

Ensuite, ajoutons les attributs de la classe ; mais pour cela, réfléchissons les données utiles pour notre combat.
Comme dans tous les jeux avec des combats (ou presque), donnons un niveau de santé à nos personnages, que nous initialiserons à 100. Et pour les armer, indiquons la puissance de l'attaque qu'ils vont porter à leur adversaire, ainsi que les chances de coup critique :

Code : Actionscript

```
// Santé du personnage
private var _sante:int;

// Dégâts de base
private var _degats:int;

// Chances de faire une critique (sur 100)
private var _chanceCritique:int;
```

Les accesseurs

N'oublions pas d'accompagner les attributs de leurs accesseurs :

Code : Actionscript

```
public function get sante():int
{
 return _sante;
}

public function set sante(value:int):void
{
 _sante = value;
}

public function get degats():int
{
 return _degats;
}

public function set degats(value:int):void
{
 _degats = value;
}

public function get chanceCritique():int
{
 return _chanceCritique;
}

public function set chanceCritique(value:int):void
{
 _chanceCritique = value;
}
```

Le contructeur

Ensuite, initialisons nos attributs au sein du constructeur :

Code : Actionscript

```
// Constructeur
public function Personnage()
{
 sante = 100;
 degats = 0;
 chanceCritique = 0;
}
```

La méthode

Enfin, ajoutons une méthode, afin que nos personnages puissent attaquer un autre personnage :

Code : Actionscript

```
public function attaquer(cible:Personnage):void
{
 var degatsAppliques:int = degats;

 // On jette un dé à 100 faces : si le résultat est inférieur ou
égal à la chance de coup critique, l'attaque fait un coup critique
!
 if (Math.random() * 100 <= chanceCritique)
 {
  trace("Critique !");
  // On double les dégâts !
  degatsAppliques *= 2;
 }

 // On applique les dégâts
 cible.sante -= degatsAppliques;

 if (cible.sante <= 0)
 {
  trace("La cible est décédée.");
 }
 else
 {
  trace("Il reste " + cible.sante + " PV à la cible.");
 }
}
```

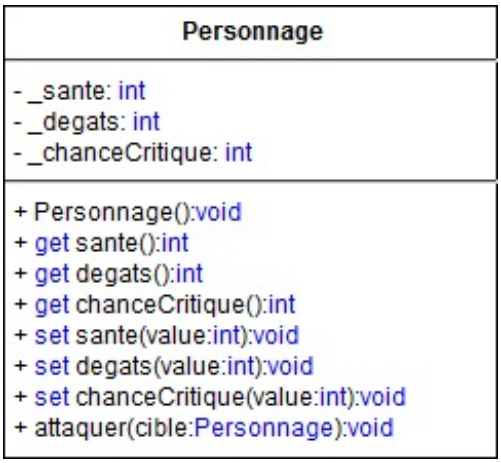
Comme vous pouvez le constater, nous passons en paramètre un objet de la classe `Personnage`, afin de rendre le code logique et surtout très lisible. Ainsi, pour qu'un personnage attaque un second, il faudra procéder ainsi :

Code : Actionscript

```
personnageA.attaquer(personnageB); // Le personnageA attaque le
personnageB ! S'en est fini de lui !
```

La classe complète

Si tout se passe bien, vous devriez normalement avoir une classe `Personnage` qui correspond à la description ci-dessous :



La classe Personnage

Voici le code complet de notre classe `Personnage`, pour vérifier le vôtre :

Code : Actionscript

```
package
```

```
{

    public class Personnage
    {

        // Santé du personnage
        private var _sante:int;

        // Dégâts de base
        private var _degats:int;

        // Chances de faire une critique (sur 100)
        private var _chanceCritique:int;

        public function Personnage()
        {
            sante = 100;
            degats = 0;
            chanceCritique = 0;
        }

        public function get sante():int
        {
            return _sante;
        }

        public function set sante(value:int):void
        {
            _sante = value;
        }

        public function get degats():int
        {
            return _degats;
        }

        public function set degats(value:int):void
        {
            _degats = value;
        }

        public function get chanceCritique():int
        {
            return _chanceCritique;
        }

        public function set chanceCritique(value:int):void
        {
            _chanceCritique = value;
        }

        public function attaquer(cible:Personnage):void
        {
            var degatsAppliques:int = degats;

            // On jette un dé à 100 faces : si le résultat est
inférieur ou égal à la chance de coup critique, l'attaque fait un
coup critique !
            if (Math.random() * 100 <= chanceCritique)
            {
                trace("Critique !");
                // On double les dégâts !
                degatsAppliques *= 2;
            }

            // On applique les dégâts
            cible.sante -= degatsAppliques;

            if (cible.sante <= 0)
            {
```

```
                    trace("La cible est décédée.");
                }
                else
                {
                    trace("Il reste " + cible.sante + " PV à la
cible.");
                }
            }
        }
}
```

Le programme principal

Votre classe `Main` vide (contenue dans le fichier `Main.as`) devrait ressembler à cela :

Code : Actionscript

```
package {
 import flash.display.Sprite;
 import flash.events.Event;

 public class Main extends Sprite {

  public function Main():void {
   if (stage)
    init();
   else
    addEventListener(Event.ADDED_TO_STAGE, init);
  }

  private function init(e:Event = null):void {
   removeEventListener(Event.ADDED_TO_STAGE, init);
   // entry point

  }
 }
}
```

Rappel : il faut commencer à programmer après le commentaire `// entry point` («point d'entrée») à la ligne 17.

Commençons par déclarer la variable qui pointera vers le premier objet de classe `Personnage` (celui qui vous représente) :

Code : Actionscript

```
var moi:Personnage = new Personnage();
```

Ensuite, donnons-lui son épée légendaire (elle fait 80 dégâts de base et a 80 chances sur 100 de faire un coup critique) :

Code : Actionscript

```
moi.degats = 80;
moi.chanceCritique = 80;
```

Le code pour créer le Méchant est très similaire :

Code : Actionscript

```
var mechant:Personnage = new Personnage();
mechant.degats = 40;
mechant.chanceCritique = 10;
```

Enfin, simulons le combat épique qui a lieu entre nos deux personnages ! Si vous vous souvenez de ma remarque sur la méthode `attaquer()` un peu plus haut, vous savez comment procéder :

Code : Actionscript

```
trace("Le méchant m'attaque ! ");
mechant.attaquer(moi);

trace("Il va connaître ma fureur ! A l'attaque !");
moi.attaquer(mechant);
```

Voici le code complet de notre classe `Main` :

Code : Actionscript

```
package
{
 import flash.display.Sprite;
 import flash.events.Event;

 public class Main extends Sprite
 {

  public function Main():void
  {
   if (stage)
    init();
   else
    addEventListener(Event.ADDED_TO_STAGE, init);
  }

  private function init(e:Event = null):void
  {
   removeEventListener(Event.ADDED_TO_STAGE, init);
   // entry point

   // Création du personnage vous représentant
   var moi:Personnage = new Personnage();
   moi.degats = 80;
   moi.chanceCritique = 80;

   // Création du personnage Méchant
   var mechant:Personnage = new Personnage();
   mechant.degats = 40;
   mechant.chanceCritique = 10;

   // Simulation du combat
   trace("Le méchant m'attaque ! ");
   mechant.attaquer(moi);
   trace("Il va connaître ma fureur ! A l'attaque !");
   moi.attaquer(mechant);
  }
 }
}
```

Résultat

Nous pouvons maintenant compiler et tester le projet. Voici ce que donne la console :

Code : Console

```
Le méchant m'attaque !
Critique !
Il reste 20 PV à la cible.
Il va connaître ma fureur ! A l'attaque !
Critique !
La cible est décédée.
```

Gagné !

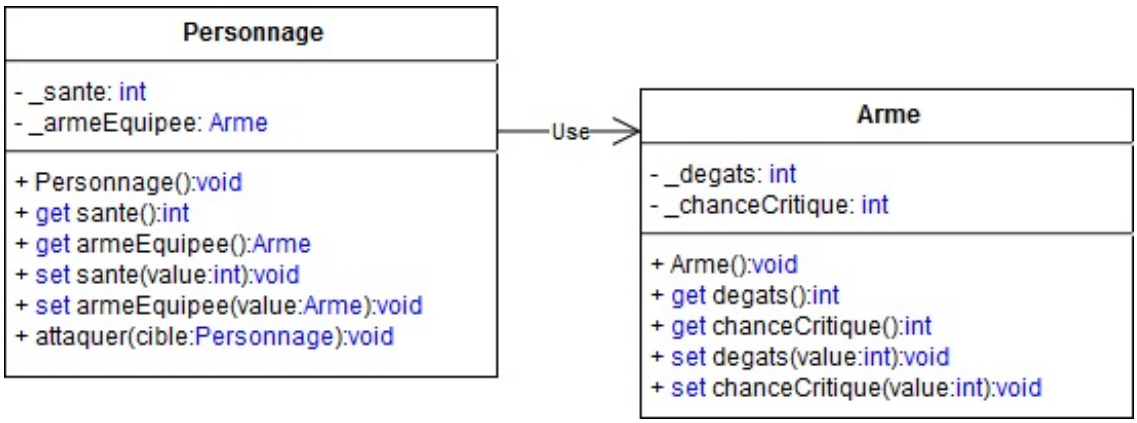
Attendez malheureux ! Ne criez pas victoire trop vite ! En effet, notre classe pourrait être améliorée...
Vous ne voyez pas en quoi ? Et bien, pensez au chapitre précédent : «Un objet dans un objet (dans un objet...)». Maintenant, réfléchissez à cette problématique : *comment pourrait-on mieux séparer les données et les propriétés de ma classe `Personnage` ?* En créant de nouvelles classes, pardi !

Une nouvelle classe

En effet, il serait judicieux de représenter les armes que portent nos personnages par des objets à part entière : cela semble logique, et cela respecte les principes de la POO. En outre, cela nous faciliterait énormément la tâche si nous devions gérer un inventaire par exemple : nous pourrions mettre autant d'objets que l'on veut, et équiper nos personnages avec, tout ceci de façon très souple et naturelle !

La classe `Arme`

L'idée est donc de transférer toutes les propriétés relatives aux armes dans une nouvelle classe `Arme`, comme ceci :



Transfert des propriétés dans la classe Arme

Il nous faudra donc créer une nouvelle classe (ici dans le fichier `Arme.as`) :

Code : Actionscript

```
package
{

    public class Arme
    {

        // Dégâts de l'arme
        private var _degats:int;

        // Chances de faire un coup critique (sur 100)
        private var _chanceCritique:int;

        public function Arme()
        {
            degats = 0;
            chanceCritique = 0;
        }
```

```
            public function get chanceCritique():int
            {
                return _chanceCritique;
            }

            public function set chanceCritique(value:int):void
            {
                _chanceCritique = value;
            }

            public function get degats():int
            {
                return _degats;
            }

            public function set degats(value:int):void
            {
                _degats = value;
            }
        }
    }
```

La classe `Personnage`

N'oublions pas d'adapter la classe `Personnage`, comme nous l'avons fait dans le chapitre précédent :

Code : Actionscript

```
    package
    {

        public class Personnage
        {

            // Santé du personnage
            private var _sante:int;

            // Arme équipée
            private var _armeEquipee:Arme; // Nouvel attribut pointant
    sur l'arme équipée

            public function Personnage()
            {
                sante = 100;
            }

            public function get sante():int
            {
                return _sante;
            }

            public function set sante(value:int):void
            {
                _sante = value;
            }

            public function get armeEquipee():Arme // Nouvel accesseur
            {
                return _armeEquipee;
            }

            public function set armeEquipee(value:Arme):void // Nouveau
    mutateur
            {
                _armeEquipee = value;
            }
```

```
        public function attaquer(cible:Personnage):void
        {
            // Au cas où aucun arme n'est équipée (l'objet
armeEquipee est null)
            if (armeEquipee == null)
            {
                trace("Aucune arme équipée : l'attaque échoue.");
            }
            else
            {
                var degatsAppliques:int = armeEquipee.degats; //
Désormais, on utilise les propriétés de l'objet armeEquipee

                if (Math.random() * 100 <=
armeEquipee.chanceCritique) // Ici aussi, on utilise les propriétés
de l'objet armeEquipee
                {
                    trace("Critique !");
                    // On double les dégâts !
                    degatsAppliques *= 2;
                }

                // On applique les dégâts
                cible.sante -= degatsAppliques;

                if (cible.sante <= 0)
                {
                    trace("La cible est décédée.");
                }
                else
                {
                    trace("Il reste " + cible.sante + " PV à la
cible.");
                }
            }
        }
    }
}
```

Le programme principal

Là aussi, il va falloir adapter un peu : au lieu d'affecter directement les dégâts et les chances de critique aux personnages, nous créons dans un premier temps les armes via des objets de classe `Arme`, pour ensuite les équiper aux personnages :

Code : Actionscript

```
var epeeLegendaire:Arme = new Arme();
epeeLegendaire.degats = 80;
epeeLegendaire.chanceCritique = 50;

var hacheDeGuerre:Arme = new Arme();
hacheDeGuerre.degats = 40;
hacheDeGuerre.chanceCritique = 10;

var moi:Personnage = new Personnage();
moi.armeEquipee = epeeLegendaire;

var mechant:Personnage = new Personnage();
mechant.armeEquipee = hacheDeGuerre;

trace("Le méchant m'attaque ! ");
mechant.attaquer(moi);
trace("Il va connaître ma fureur ! A l'attaque !");
moi.attaquer(mechant);
```

Avouez, le code est quand même plus clair que le précédent !

Résultat

Et voici le résultat à la console lorsque l'on teste le projet :

Code : Console

```
Le méchant m'attaque !
Il reste 60 PV à la cible.
Il va connaître ma fureur ! A l'attaque !
Critique !
La cible est décédée.
```

Rien n'a vraiment changé (à part ma chance qui s'est envolée) : ce n'est toutefois pas pour rien que nous avons modifié notre code. En effet, *il est primordial de programmer correctement pour que vos projets soient lisibles, facilement modifiables et maintenables.*

Malheureusement, notre code pose encore problème : il ne respecte pas bien le principe d'**encapsulation**. Si vous regardez bien la méthode `attaquer()`, nous utilisons des propriétés de la classe `Arme` et reproduisons son comportement (à savoir : les coups critiques) dans la classe `Personnage` : en toute logique, si une arme devrait faire un coup critique, nous devrions le déterminer dans la bonne classe, autrement dit la classe `Arme` !

La bonne solution

La bonne façon de procéder consiste donc à appliquer les dégâts qui vont être fait dans la classe `Arme`. Pour cela, créons dans cette classe une nouvelle méthode `frapper()` :

Code : Actionscript

```
public function frapper(cible:Personnage):void
{
 var degatsAppliques:int = degats;

 // On jette un dé à 100 faces : si le résultat est inférieur ou
égal à la chance de coup critique, l'attaque fait un coup critique
!
 if (Math.random() * 100 <= chanceCritique)
 {
  trace("Critique !");
  // On double les dégâts !
  degatsAppliques *= 2;
 }

 // On applique les dégâts
 cible.sante -= degatsAppliques;
}
```

Il va donc falloir appeler cette nouvelle méthode dans la méthode `attaquer()` de la classe `Personnage` :

Code : Actionscript

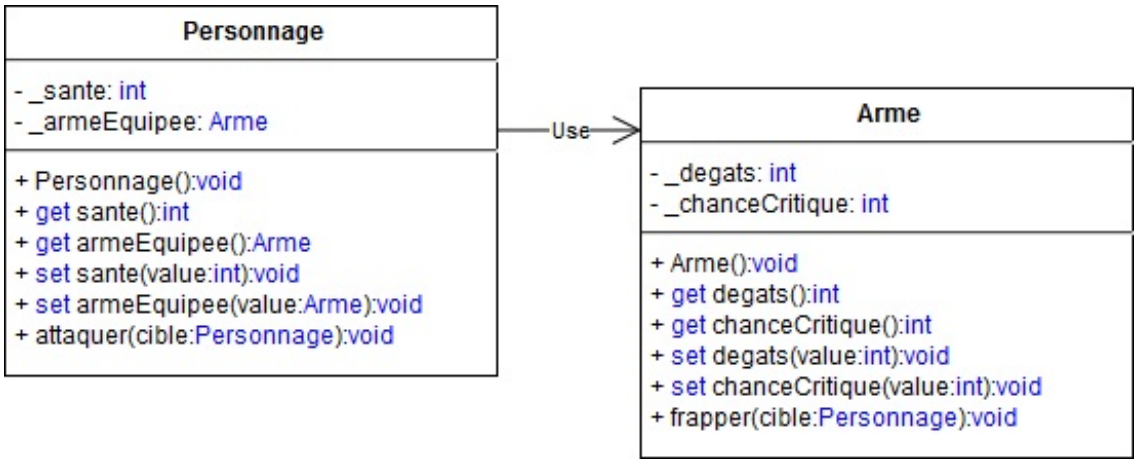
```
public function attaquer(cible:Personnage):void
{
 if (armeEquipee == null)
 {
  trace("Aucune arme équipée : l'attaque échoue.");
 }
 else
 {
```

```
        armeEquipee.frapper(cible); // Nous appelons la nouvelle méthode
    ici

        if (cible.sante <= 0)
        {
         trace("La cible est décédée.");
        }
        else
        {
         trace("Il reste " + cible.sante + " PV à la cible.");
        }
       }
    }
```

Les classes

L'un des intérêts de l'utilisation de la représentation UML est justement de faciliter cette étape de conception et d'organisation des différentes classes d'un même programme. Cela permet de visualiser la structure d'un projet en ne faisant ressortir que les informations utiles, et ainsi programmer plus rapidement et de manière plus propre.

Au final, vos classes devraient ressembler à ceci :



Les classes Personnage et Arme

Voici le code complet de nos classes :

Code : Actionscript

```
package
{

    public class Personnage
    {

        // Santé du personnage
        private var _sante:int;

        // Amre équipée
        private var _armeEquipee:Arme;

        public function Personnage()
        {
            sante = 100;
        }

        public function get sante():int
        {
            return _sante;
        }
```

```
        public function set sante(value:int):void
        {
            _sante = value;
        }

        public function get armeEquipee():Arme
        {
            return _armeEquipee;
        }

        public function set armeEquipee(value:Arme):void
        {
            _armeEquipee = value;
        }

        public function attaquer(cible:Personnage):void
        {
            if (armeEquipee == null)
            {
                trace("Aucune arme équipée : l'attaque échoue.");
            }
            else
            {
                armeEquipee.frapper(cible);

                if (cible.sante <= 0)
                {
                    trace("La cible est décédée.");
                }
                else
                {
                    trace("Il reste " + cible.sante + " PV à la
cible.");
                }
            }
        }
    }
}
```

Code : Actionscript

```
package
{

    public class Arme
    {

        // Dégâts de l'arme
        private var _degats:int;

        // Chances de faire un coup critique (sur 100)
        private var _chanceCritique:int;

        public function Arme()
        {
            degats = 0;
            chanceCritique = 0;
        }

        public function get chanceCritique():int
        {
            return _chanceCritique;
        }

        public function set chanceCritique(value:int):void
        {
            _chanceCritique = value;
```

```
            }

            public function get degats():int
            {
                return _degats;
            }

            public function set degats(value:int):void
            {
                _degats = value;
            }

            public function frapper(cible:Personnage):void
            {
                var degatsAppliques:int = degats;

                // On jette un dé à 100 faces : si le résultat est
    inférieur ou égal à la chance de coup critique, l'attaque fait un
    coup critique !
                if (Math.random() * 100 <= chanceCritique)
                {
                    trace("Critique !");
                    // On double les dégâts !
                    degatsAppliques *= 2;
                }

                // On applique les dégâts
                cible.sante -= degatsAppliques;
            }
        }
    }
```

Le programme

Le programme principal ne change pas par rapport à la solution précédente :

Code : Actionscript

```
var epeeLegendaire:Arme = new Arme();
epeeLegendaire.degats = 80;
epeeLegendaire.chanceCritique = 50;

var hacheDeGuerre:Arme = new Arme();
hacheDeGuerre.degats = 40;
hacheDeGuerre.chanceCritique = 10;

var moi:Personnage = new Personnage();
moi.armeEquipee = epeeLegendaire;

var mechant:Personnage = new Personnage();
mechant.armeEquipee = hacheDeGuerre;

trace("Le méchant m'attaque ! ");
mechant.attaquer(moi);
trace("Il va connaître ma fureur ! A l'attaque !");
moi.attaquer(mechant);
```

Résultat

Enfin, voici le résultat de l'exécution de notre programme :

Code : Console

```
Le méchant m'attaque !
Il reste 60 PV à la cible.
Il va connaître ma fureur ! A l'attaque !
Il reste 20 PV à la cible.
```

En résumé

- Il est possible d'utiliser des éléments dits **statiques** qui sont directement liés à la classe et non à ses instances.
- Ces éléments statiques sont déclarés à l'aide du mot-clé `static` et facilitent l'ajout de fonctionnalités au programme.
- Il est impossible d'utiliser le mot-clé `this` dans les méthodes statiques.
- Nous pouvons créer des **constantes** qui sont similaires aux variables, mais qui ne peuvent pas être modifiées.
- Pour déclarer une constante, nous devons utiliser le mot-clé `const`.
- Il est possible de **combiner** les classes entre elles : les objets peuvent alors contenir d'autres objets.

L'héritage

Dans ce chapitre, nous allons parler d'**héritage** !
Sachez qu'il s'agit de l'une des notions les plus importantes de la programmation orientée objet, et qui en font tout son intérêt !
Nous détaillerons ce concept tout au long du chapitre, mais pour vous remettre les idées en place, cela permet de créer une ou des nouvelles classes en se basant sur une autre. Entre autres, cela permet d'écrire des classes qui sont similaires en utilisant une autre classe qui regroupe l'ensemble des propriétés communes.
Dans ce chapitre, la seule difficulté présente est la notion d'*héritage* en elle-même. Au niveau du code, il n'y aura pas énormément de nouveautés.

La notion d'héritage

Nous avons déjà brièvement présenté le concept, cependant vu son importance, ce n'est pas superflu d'en remettre une couche !
L'**héritage** permet de créer une ou des nouvelles classes en réutilisant le code d'une classe déjà existante. On parle alors de *classe de base* ou **superclasse** ou encore **classe mère** pour cette dernière, et de **sous-classes** ou **classes filles** pour les classes *héritées* de celle-ci. L'idée est donc ici d'*étendre* une classe de base, notamment en lui ajoutant de nouvelles propriétés.
D'ailleurs le mot-clé qui permet d'étendre une classe est `extends`, que nous avons déjà croisé dans le deuxième chapitre:

Code : Actionscript

```
public class Hello extends Sprite {

}
```

Nous reviendrons sur la syntaxe Actionscript plus tard. Pour l'instant, nous allons principalement nous focaliser sur la notion d'héritage.

Quand est-ce utile d'utiliser l'héritage ?

C'est en effet une question à laquelle les débutants ont souvent du mal à répondre. En réalité, nous pouvons introduire une relation d'héritage lorsque la condition suivante est respectée :
*la sous-classe est un **sous-ensemble** de la superclasse.*

Ce terme mathématique barbare signifie que la sous-classe appartient à l'ensemble de la superclasse. Si ce n'est pas encore bien clair, voici quelques exemples qui vous aideront à bien comprendre :

- l'Actionscript appartient à l'ensemble des langages de programmation
- les fourmis appartiennent à l'ensemble des insectes
- les avions appartiennent à l'ensemble des véhicules
- les voitures appartiennent également à l'ensemble des véhicules
- les 4L appartiennent à l'ensemble des voitures.

Comme vous pouvez le constater, les relations précédentes ne peuvent s'effectuer que dans *un seul sens*. Vous remarquerez également d'après les deux derniers exemples qu'il est possible d'avoir *plusieurs niveaux* d'héritage.

En quoi est-ce différent d'une instance de classe ?

Il est vrai qu'il serait possible par exemple, de créer des instances `Avion` et `Voiture` d'une classe `Vehicule`. Nous pourrions de cette manière définir des valeurs distinctes pour chacun des attributs afin de les différencier.
En utilisant le concept d'héritage, nous pourrions écrire deux sous-classes `Avion` et `Voiture` qui hériteraient de l'ensemble des attributs et méthodes de la superclasse, et ce sans réécrire le code à l'intérieur de celles-ci. Mais tout l'intérêt vient du fait que l'utilisation de l'héritage nous permet de définir de nouveaux attributs et de nouvelles méthodes pour nos sous-classes.
Sachez qu'il est également possible de redéfinir des méthodes de la superclasse, mais nous en reparlerons quand nous introduirons le **polymorphisme** plus loin dans ce chapitre !

Construction d'un héritage

Dans la suite de ce chapitre, nous allons principalement nous intéresser aux manipulations du côté des classes filles. Cependant nous aurons besoin d'une superclasse à partir de laquelle nous pourrons travailler. C'est pourquoi je vous propose une classe `Vehicule`, dont le code est donné ci-dessous :

Code : Actionscript

```
package
{
    public class Vehicule
    {
        protected var _marque:String;
        protected var _vitesse:int;

        public function Vehicule(marque:String, vitesse:int)
        {
            _marque = marque;
            _vitesse = vitesse;
        }

        public function get marque():String
        {
            return _marque;
        }

        public function get vitesse():int
        {
            return _vitesse;
        }

        public function set vitesse(vitesse:int):void
        {
            _vitesse = vitesse;
        }

    }

}
```

La portée `protected`

Nous avions rapidement mentionné cette portée sans en expliquer vraiment le fonctionnement. Maintenant vous savez comment fonctionne l'héritage, nous allons pouvoir utiliser cette nouvelle portée qu'est **`protected`** !
Cette portée a été introduite afin de rendre les propriétés visibles non seulement depuis la classe où elles sont définies comme **`private`**, mais également depuis l'ensemble de ses sous-classes.
Voyez les attributs de la classe `Vehicule` définie juste avant :

Code : Actionscript

```
protected var _marque:String;
protected var _vitesse:int;
```

Ces attributs seront donc visibles depuis les éventuelles sous-classes que nous pourrons définir.

Lorsque nous avions introduit le concept d'encapsulation, nous avions dit qu'il fallait spécifier une portée de type **`private`** à tous vos attributs. Or comme nous l'avons dit, les attributs de ce type ne sont accessibles que depuis la classe où ils ont été déclarés. C'est pourquoi il est maintenant préférable d'opter pour la portée **`protected`** à chaque fois que l'une de vos classes est susceptible devenir une *superclasse*. En pratique, on utilise quasiment tout le temps cette portée dans l'hypothèse d'un héritage futur.

À présent vous connaissez l'ensemble des portées que propose l'Actionscript, dont voici un petit récapitulatif :

- **`public`** : propriété visible n'importe où
- **`private`** : propriété visible uniquement à l'intérieur de la classe qui l'a définie
- **`protected`** : propriété visible depuis la classe où elle est définie, ainsi que depuis l'ensemble de ses sous-classes
- **`internal`** : propriété visible depuis l'ensemble du package où elle est définie.

Construction des sous-classes

L'héritage

Comme nous l'avons dit en début de chapitre, l'héritage se fait en « étendant » une classe à l'aide du mot-clé **extends** comme ceci :

Code : Actionscript

```
package
{
    public class Voiture extends Vehicule
    {

    }
}
```

Attention à l'ordre dans lequel sont placés les différents éléments. Nous sommes ici en train de déclarer une nouvelle classe nommée `Voiture` qui hérite de la classe `Vehicule`.

Par cette simple extension, la classe `Voiture` hérite donc de l'ensemble des propriétés de la classe `Vehicule`. Il est toutefois nécessaire de lui rajouter au moins un constructeur, et éventuellement quelques propriétés supplémentaires pour lui donner de l'intérêt.
Par exemple, nous pourrions introduire un attribut `_traction` pour définir si la voiture est une traction ou non, ou encore un attribut `_immatriculation` pour identifier celle-ci :

Code : Actionscript

```
package
{
    public class Voiture extends Vehicule
    {
        protected var _traction:Boolean;
        protected var _immatriculation:String;
    }
}
```

Le constructeur

Comme toute classe, notre sous-classe `Voiture` doit posséder un constructeur. Cependant, celle-ci hérite d'une classe mère qui possède son propre constructeur qui est nécessaire à l'initialisation des attributs. Depuis une classe fille, il est possible d'appeler le constructeur de sa superclasse par la fonction **super**`()` :

Code : Actionscript

```
public function Voiture()
{
    super();
    // Instructions supplémentaires
}
```

L'appel du constructeur **super**`()` de la superclasse doit *obligatoirement* être la première instruction du constructeur de votre sous-classe. Les nouvelles instructions seront placées à la suite pour permettre l'initialisation de vos nouvelles variables.

Voici donc un exemple de constructeur pour notre classe `Voiture` :

Code : Actionscript

```
public function Voiture(marque:String, vitesse:int,
immatriculation:String)
{
    super(marque, vitesse);
    _immatriculation = immatriculation;
    _traction = true;
}
```

Vous remarquerez que la fonction `super()` est le constructeur de la *superclasse*. Il est donc normal de retrouver les différents paramètres du constructeur défini plus haut.

Les méthodes

En plus de pouvoir définir de nouveaux attributs, il est possible de rajouter autant de méthodes que nous souhaitons à l'intérieur d'une sous-classe. Étant donné nous utilisons encore le concept d'encapsulation, nous commencerons par créer des accesseurs à ce nouvel attribut. Je vous propose de découvrir quelques-uns d'entre eux :

Code : Actionscript

```
public function set
immatriculation(nouvelleImmatriculation:String):void
{
    _immatriculation = nouvelleImmatriculation;
}

public function get immatriculation():String
{
    return _immatriculation;
}
```

Comme vous pouvez le voir, ces méthodes fonctionnent exactement de la même manière que pour une classe quelconque. En revanche ce qu'il peut être intéressant, c'est d'utiliser les méthodes définies à l'intérieur de la classe mère. Ainsi si les méthodes de votre classe mère ont une portée **public** ou **protected**, celles-ci sont accessibles depuis les classes filles. Nous avons ainsi un mot-clé **super** qui nous permet de faire référence à la superclasse, et d'utiliser ses propriétés.
Voici par exemple une méthode nommée `accelerer()` qui permet d'augmenter la vitesse du véhicule :

Code : Actionscript

```
public function accelerer():void
{
    var nouvelleVitesse:int = super.vitesse + 15;
    super.vitesse = nouvelleVitesse;
}
```

Vous noterez que nous aurions pu utiliser directement l'attribut `_vitesse` de la classe `Vehicule` pour définir la variable `nouvelleVitesse`. En revanche nous sommes obligés d'utiliser l'accesseur `vitesse()` pour modifier la valeur de l'attribut. En effet, seules les propriétés définies par le mot-clé **function**, donc les méthodes, peuvent être redéfinies. Nous reviendrons là-dessus juste après, lorsque nous parlerons du *polymorphisme*.

Le mot-clé **super** fait référence à l'objet via la classe mère, par opposition au mot-clé **this** qui pointe sur l'objet en lui-même .

Néanmoins, le mot-clé **super** est facultatif dans la plupart des cas. Le code ci-dessous est tout à fait fonctionnel :

Code : Actionscript

```
public function accelerer():void
{
    var nouvelleVitesse:int = vitesse + 15; // L'accesseur de la
classe-mère sera automatiquement sélectionné
    vitesse = nouvelleVitesse;
}
```

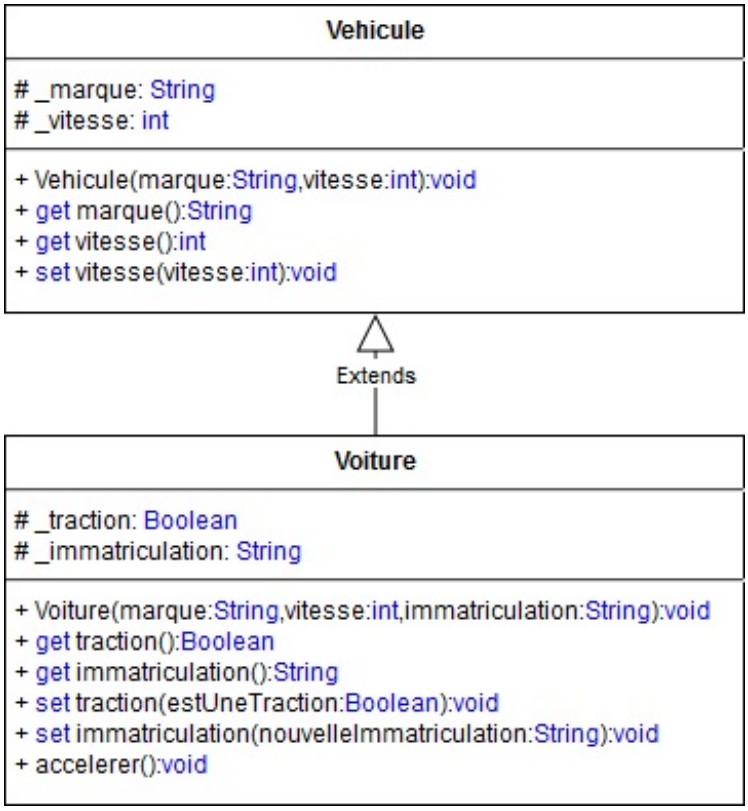
Il peut même être simplifié (vu que vous avez compris le principe) :

Code : Actionscript

```
public function accelerer():void
{
    vitesse += 15;
}
```

Rappel : les accesseurs simulent le fonctionnement des attributs. Il est donc possible d'utiliser ici tous les opérateurs mathématiques : `vitesse` est considéré comme un nombre.

Comme je n'ai pas tout réécrit, je vous propose tout de même un schéma UML résumant les propriétés des deux classes `Vehicule` et `Voiture` ainsi que le lien qui les unit :



Héritage entre les classes Vehicule et Voiture

La substitution d'une sous-classe à une superclasse

Un autre avantage de l'utilisation de l'héritage est le fait de pouvoir *substituer une sous-classe à une superclasse*. C'est-à-dire qu'il est possible de manipuler une classe fille comme s'il s'agissait d'une instance de la classe mère.

Parce qu'un exemple vaut mille mots, prenons le code suivant :

Code : Actionscript

```
var MesVehicules:Array = new Array();
```

```
MesVehicules.push(new Vehicule("Airbus A380", 900));
MesVehicules.push(new Vehicule("Bicyclette", 25));
MesVehicules.push(new Voiture("Renault 4L", 100, "911 SDZ 15"));
for (var i:int = 0; i < MesVehicules.length; i++)
{
    trace("Un véhicule de type " + MesVehicules[i].marque + " peut
se déplacer à la vitesse de " + MesVehicules[i].vitesse + "km/h.");
}
/* Affiche :
Un véhicule de type Airbus A380 peut se déplacer à la vitesse de
900km/h.
Un véhicule de type Bicyclette peut se déplacer à la vitesse de
25km/h.
Un véhicule de type Renault 4L peut se déplacer à la vitesse de
100km/h.
*/
```

Il n'y a ici rien de surprenant, les accesseurs de la classe `Vehicule` sont bien accessibles depuis la classe `Voiture`. En revanche ce qui deviendrait intéressant, ce serait de créer une méthode `presenter()` qui permet de présenter un objet de type `Vehicule`, comme ci-dessous :

Code : Actionscript

```
var MesVehicules:Array = new Array();
MesVehicules.push(new Vehicule("Airbus A380", 900));
MesVehicules.push(new Vehicule("Bicyclette", 25));
MesVehicules.push(new Voiture("Renault 4L", 100, "911 SDZ 15"));
function presenter(unVehicule:Vehicule):void
{
    trace("Un véhicule de type " + unVehicule.marque + " peut se
déplacer à la vitesse de " + unVehicule.vitesse + "km/h.");
}
for (var i:int = 0; i < MesVehicules.length; i++)
{
    presenter(MesVehicules[i]);
}
/* Affiche :
Un véhicule de type Airbus A380 peut se déplacer à la vitesse de
900km/h.
Un véhicule de type Bicyclette peut se déplacer à la vitesse de
25km/h.
Un véhicule de type Renault 4L peut se déplacer à la vitesse de
100km/h.
*/
```

Comment se fait-il qu'il n'y ait pas d'erreur pour l'objet de type `Voiture` ?

Comme nous l'avons dit, nous pouvons *substituer une sous-classe à une superclasse*. En d'autres termes, il est possible d'utiliser une classe fille comme s'il s'agissait de la classe mère. D'ailleurs si vous vous rappelez bien nous avions dit qu'une sous-classe était un *sous-ensemble* de la superclasse, ce qui veut dire qu'une `Voiture` *est* un `Vehicule`. Il n'est donc pas surprenant de pouvoir utiliser un objet de type `Voiture` en tant que `Vehicule`.

Encore une fois, attention au sens de l'héritage !
Dans notre exemple, il n'est pas possible de substituer un `Vehicule` à une `Voiture` ; seul le sens opposé est exact !

Le polymorphisme

Nous allons maintenant parler du **polymorphisme** !
Nous avons là-encore un nom barbare associé à un concept qui n'est très compliqué au fond.

Précédemment, nous avons appris à étendre une superclasse en ajoutant de nouvelles méthodes à l'intérieur d'une sous-classe. Cependant il est également possible de *redéfinir* (réécrire) une méthode. Ainsi nous avons la possibilité d'utiliser un nom de méthode commun pour une méthode qui se comportera différemment suivant le type de l'objet.

Pour vous montrer cela, nous allons insérer une nouvelle méthode qu'on nommera `sePresenter()` à l'intérieur de la classe `Vehicule` :

Code : Actionscript

```
public function sePresenter():void
{
    trace("Un véhicule de type " + marque + " peut se déplacer à la
vitesse de " + vitesse + "km/h.");
}
```

Rappel : il est conseillé d'utiliser les accesseurs dans la classe elle-même (sauf dans le constructeur).

Si nous ne faisons rien, la classe `Voiture` héritera de cette nouvelle méthode, et nous pourrons l'utiliser sans problème. Cependant nous aimerions personnaliser le message, notamment en rajoutant son numéro d'immatriculation qui permet également de l'identifier.
Nous devrons donc réécrire la méthode `sePresenter()` pour la classe `Voiture`. Heureusement nous disposons d'un mot-clé **`override`**, qui permet justement de redéfinir une méthode. Voici comment l'utiliser :

Code : Actionscript

```
override public function sePresenter():void
{
    trace("Une voiture " + marque + " peut se déplacer à la vitesse
de " + vitesse + "km/h.");
    trace("Son immatriculation est : " + immatriculation);
}
```

Ainsi nous pouvons utiliser la méthode `sePresenter()` sans nous soucier du type d'objets que nous sommes en train de manipuler.

Code : Actionscript

```
var MesVehicules:Array = new Array();
MesVehicules.push(new Vehicule("Airbus A380", 900));
MesVehicules.push(new Vehicule("Bicyclette", 25));
MesVehicules.push(new Voiture("Renault 4L", 100, "911 SDZ 75"));
for (var i:int = 0; i < MesVehicules.length; i++)
{
    MesVehicules[i].sePresenter();
}
/* Affiche :
Un véhicule de type Airbus A380 peut se déplacer à la vitesse de
900km/h.
Un véhicule de type Bicyclette peut se déplacer à la vitesse de
25km/h.
Une voiture Renault 4L peut se déplacer à la vitesse de 100km/h.
Son immatriculation est : 911 SDZ 75
*/
```

Pour résumer, le polymorphisme est une technique très puissante, surtout lorsqu'elle est associée à la substitution d'une sous-classe à une superclasse. Nous pouvons ainsi définir des méthodes « par défaut » dans la classe mère, puis de les redéfinir pour différentes classes filles. Il est ensuite possible d'utiliser ces différents objets *de la même façon* comme s'il s'agissait de la même classe. Vous verrez dans la suite que c'est un atout non négligeable !

Les attributs de classe

Lorsque nous avions parlé d'*encapsulation*, nous avions introduit les droits d'accès pour les différentes propriétés d'une classe. Or pour ceux qui l'auraient également remarqué, nous avons depuis le départ toujours inséré le mot-clé **`public`** devant la définition de chacune de nos classes.

Il existe aussi des droits d'accès pour les classes ?

En effet, tout comme les propriétés, les classes possèdent des droits d'accès qui permettent de définir comment nous pouvons accéder à la classe et même la modifier. En réalité, on parle d'**attributs de classes** et d'**attributs de propriétés de classes**, mais nous emploierons dans ce cours le terme « droits d'accès » pour éviter la confusion avec les variables internes aux classes appelées également *attributs*.

Les différents droits d'accès

En ce qui concerne la définition de classes, l'Actionscript propose quatre droits d'accès différents. Sans plus attendre, je vous propose de les découvrir :

- **`public`** : les droits d'accès « publiques » permettent comme pour les propriétés, de rendre la classe visible et accessible partout dans le code. Il s'agit des droits d'accès recommandés dans la majorité des cas.
- **`internal`** : identiquement aux propriétés, les droits d'accès « internes » restreignent l'accessibilité de la classe au package où elle est définie uniquement. Également ces droits d'accès ne sont pas très utilisés, mais il s'agit de la valeur par défaut lors d'une définition de classe.
- **`final`** : ces droits d'accès sont directement liés à la notion d'héritage. Le terme « final » fait ici référence au fait que la classe ne peut plus être étendue par une autre classe.
- **`dynamic`** : ce mot-clé permet de définir une classe **dynamique**, c'est-à-dire modifiable depuis l'extérieur de celle-ci, à l'opposé des classes classiques dites **scellées**. Nous reviendrons sur ce concept un peu particulier dans le prochain chapitre.

Pour résumer tout ce qui concerne les droits d'accès et l'encapsulation, vous devez vous rappeler qu'on doit principalement limiter l'accès aux attributs en utilisant préférentiellement le mot-clé **`protected`**. Pour les méthodes et les classes en général, vous privilégierez principalement un accès « publique » à l'aide du mot-clé **`public`**.
Il existe néanmoins divers autres droits d'accès, qui ne sont utiles que dans de rares occasions.

Exemple d'utilisation

Vous l'aurez compris, ces droits d'accès s'utilisent également devant la déclaration de la classe en question. Voici par exemple la définition de la classe `Vehicule` que nous avons réalisée au début du chapitre :

Code : Actionscript

```
public class Vehicule
{

}
```

Nous allons essayer ici de comprendre un peu mieux l'utilité du mot-clé **`final`**, étroitement lié au concept d'héritage !
Ce mot-clé permet de définir la classe à laquelle il est associé, comme étant la dernière classe qui finalise l'arborescence d'héritage. C'est-à-dire que cette classe peut très bien hériter d'une autre classe, mais en aucun cas vous ne pourrez créer de classes filles à celle-ci.
Je vous propose donc une petite manipulation afin de vérifier la véracité de ces propos. Redéfinissez donc la classe `Vehicule` de type **`final`** :

Code : Actionscript

```
package
{
    final class Vehicule
    {
        protected var _marque:String;
        protected var _vitesse:int;

        public function Vehicule(marque:String, vitesse:int)
        {
```

```
                _marque = marque;
                _vitesse = vitesse;
            }

            public function get marque():String
            {
                return _marque;
            }

            public function get vitesse():int
            {
                return _vitesse;
            }

            public function set vitesse(vitesse:int):void
            {
                _vitesse = vitesse;
            }

        }

    }
```

Nous avons donc maintenant une classe `Vehicule` qu'il nous est interdit d'étendre. Voyons donc ce qui se passe lorsqu'on tente d'en créer une sous-classe :

Code : Actionscript

```
package
{
    public class Voiture extends Vehicule
    {

    }
}
```

Si vous tentez donc de lancer votre programme, le compilateur refusera de compiler le projet en vous précisant l'erreur suivante : « Base class is final. », qui signifie que la classe dont on essaie d'hériter est de type **`final`**, et que notre héritage est donc contraire à cette définition de classe.

Comme vous pouvez le constater, ce mot-clé **`final`** n'apporte aucune réelle fonctionnalité, mais il s'agit plutôt d'une sécurité. Néanmoins l'utilité de ce type de droits d'accès est assez restreinte, et vous définirez majoritairement des classes de type **`public`**.

En résumé

- L'**héritage** permet de créer une ou des nouvelles classes en utilisant le code d'une classe déjà existante.
- Pour hériter d'une classe, il faut utiliser le mot-clé **`extends`**.
- La portée **`protected`** est spécifique à l'héritage.
- Le constructeur de la **superclasse** peut être appelé par la fonction **`super`**`()`.
- Dans le cas d'une relation par héritage, il est possible de *substituer* une **sous-classe** à une superclasse.
- Le **polymorphisme** permet de **redéfinir** une méthode de la superclasse par l'intermédiaire du mot-clé **`override`**.
- Les différents droits d'accès liés à la définition d'une classe sont **`public`**, **`internal`**, **`final`** et **`dynamic`**.

Notions avancées de la POO

Nous allons maintenant découvrir des notions disons plus… « avancées » de la POO. Avant de démarrer ce chapitre, laissez-moi vous préciser que les concepts abordés ici sont généralement réservés aux projets complexes ou d'envergure. Il est donc fort probable que vous n'utilisiez pas ces concepts dans vos premiers projets et même que vous n'en voyez pas encore l'utilité. C'est pourquoi il n'est pas essentiel pour la suite du cours, d'être totalement au clair avec ces notions. Lisez donc attentivement ce chapitre, mais n'hésitez pas à y revenir plus tard pour effectuer une lecture plus approfondie.

Les classes dynamiques

Définition de la classe de base

Nous allons ici revenir sur la notion de **classes dynamiques** présentée dans le chapitre précédent. Contrairement aux classes normales dites « scellées », le principe des classes dynamiques est de pouvoir rajouter de nouvelles propriétés lors de l'exécution du programme.
Bien évidemment avant d'ajouter de nouvelles propriétés, il est d'abord nécessaire de définir une classe de base, mais cette fois à l'aide du type `dynamic`.

Nous allons donc partir d'une classe `Personnage` très basique :

Code : Actionscript

```
package
{

    dynamic class Personnage
    {

        protected var _sante:int;

        public function Personnage()
        {
            sante = 100;
        }

        public function get sante():int
        {
            return _sante;
        }

        public function set sante(value:int):void
        {
            _sante = value;
        }
    }
}
```

Nous en reparlerons un peu plus loin, mais notez toutefois qu'une classe dynamique respecte le principe d'encapsulation. Nous avons ainsi nos attributs qui sont restreints à la classe ainsi qu'à d'éventuelles sous-classes.

Nous pouvons ainsi instancier autant de fois qu'on veut cette classe `Personnage`, comme ci-dessous :

Code : Actionscript

```
var guerrier:Personnage = new Personnage();
var magicien:Personnage = new Personnage();
```

Nous allons maintenant voir comment étendre les fonctionnalités de notre classe durant l'exécution en ajoutant de nouvelles propriétés à celle-ci.

Définition de propriétés hors de la classe

En pratique

Maintenant que notre classe `Personnage` est définie de type **`dynamic`**, nous allons pouvoir créer une instance de celle-ci, puis lui ajouter de nouvelles propriétés.
Comme vous avez pu le deviner d'après la phrase précédente, la définition de nouvelles propriétés se fait à partir d'une *occurrence* de la classe en question. Nous allons donc prendre l'exemple d'un `magicien` :

Code : Actionscript

```
var magicien:Personnage = new Personnage();
```

Ce type de personnage pourrait donc être doté de facultés magiques. Nous allons donc lui rajouter un nouvel attribut `_mana` qui définit la réserve de magie du personnage :

Code : Actionscript

```
magicien._mana = 50;
```

Nous avons ainsi créé ici un nouvel attribut nommé `_mana`. Cette nouvelle propriété sera alors effective *à l'exécution* du code, et sera liée *uniquement* à l'occurrence `magicien` de la classe `Personnage`. C'est pourquoi un nouvel objet `guerrier` de cette classe, ne disposerait pas de cet attribut.

L'utilisation de classes dynamiques ne se limite pas simplement à la définition de nouveaux attributs, mais s'étend également la définition de nouvelles méthodes.
Définissons par exemple une nouvelle méthode `lancerSort()` spécifique à l'instance `magicien` de la classe `Personnage` :

Code : Actionscript

```
magicien.lancerSort = function (cible:Personnage):void
{
    cible.sante -= 25;
    _mana -= 20;
};
```

Nous pouvons ainsi appeler cette méthode comme n'importe quelle autre de la classe de base :

Code : Actionscript

```
magicien.lancerSort(guerrier);
```

Comme vous pouvez le constater, les expressions de fonction sont particulièrement utiles pour la définition de méthodes de classes dynamiques.

Remarques importantes

Vous aurez certainement remarqué l'absence de droits d'accès liés aux nouvelles propriétés. En effet il n'est pas nécessaire de les spécifier, étant donné que ces propriétés sont liées à l'instance de la classe et non à la classe elle-même. Ainsi celles-ci ne sont accessibles qu'à partir de l'occurrence où elles ont été définies.

Pour revenir sur la notion d'encapsulation, les portées **`private`** et **`protected`** limitent l'accès des propriétés auxquelles elles sont associées *à la classe uniquement*, ou éventuellement aux classes filles. Ainsi dans le cas des classes dynamiques, la définition de nouvelles propriétés se fait à l'extérieur de la classe de base. C'est pourquoi nous n'avons pas accès aux propriétés « privées » dans la définition de ces nouvelles propriétés lors de l'exécution du code.

Pourquoi passer par des classes dynamiques alors que l'héritage permet déjà de faire tout cela ?

Il est vrai que dans la quasi-totalité des cas, vous n'aurez pas besoin de passer par la définition de classes dynamiques. D'ailleurs ces dernières ne permettent pas de séparer la définition d'une classe de son utilisation par la création d'instances. C'est pourquoi il est toujours préférable de passer par la conception de classes non dynamiques. Cependant ces classes dynamiques peuvent trouver leur utilité lorsque des propriétés dépendent de certaines données dont on ne dispose pas à la compilation.
Quoi qu'il en soit, si vous ne savez pas quelle solution utiliser, préférez l'héritage aux classes dynamiques à chaque fois que cela est possible !

Les interfaces

Problème

Introduction

Pour introduire les interfaces, nous allons nous servir d'un exemple. Nous allons reprendre l'idée de créer des personnages pour un jeu quelconque.
Pour cela, nous allons garder notre classe de base nommée `Personnage`. Cependant nous voulons dans notre jeu, non pas avoir de simples instances de la classe `Personnage`, mais plutôt des lutins, des elfes et des ogres qui héritent de cette classe de base. Ceux-ci possèdent alors des aptitudes différentes, mais qu'on pourrait classer dans deux catégories : `Magicien` et `Guerrier` !

Voici donc une présentation des différentes races :

- Lutin : ces petits bonhommes aux pouvoirs sensationnels se servent exclusivement de la magie et possèdent donc uniquement les aptitudes de la catégorie `Magicien`.
- Elfes : ces êtres baignés dans la magie, ont également un certain talent dans la conception et l'utilisation d'armes ; ils sont donc à la fois `Magicien` et `Guerrier`.
- Ogres : ces brutes tout droit sorties des grottes n'ont aucun sens de la magie, et possèdent donc seulement les aptitudes de la catégorie `Guerrier`.

Nous allons donc voir ici les problèmes auxquels nous allons nous heurter pour concevoir le code de ces différentes classes, toutefois nous verrons qu'il est possible de les contourner.

Les limites de l'héritage

L'objectif ici est donc de concevoir trois classes pour notre jeu : `Lutin`, `Elfe` et `Ogre`.
Pour réaliser cela, vous pourriez avoir l'idée d'utiliser de concept d'héritage. Nous pourrions donc organiser nos différentes classes suivant un double héritage. Ainsi nous aurions donc une classe de base `Personnage`, dont hériteraient les deux classes suivantes : `Magicien` et `Guerrier`. Puis nos trois classes `Lutin`, `Elfe` et `Ogre` seraient des classes filles à ces deux dernières.
Ceci ne poserait aucun problème en ce qui concerne les classes `Lutin` et `Ogre`. Seulement voilà, la classe Elfe devrait alors hériter en même temps des deux classes `Magicien` et `Guerrier`, or ceci est *interdit*.

En effet si dans certains langages tels que le C++ les héritages multiples sont autorisés, cette pratique est interdite en Actionscript. Ainsi une classe ne peut hériter que d'une unique superclasse !

Cependant il existe un concept en Actionscript qui permet de contourner ce problème : les **interfaces** !

Utilisation des interfaces

Le principe

Une interface n'est pas vraiment une classe, il s'agit plutôt d'une *collection de déclarations de méthodes*.
Contrairement aux classes, vous ne trouverez ni attribut ni constructeur au sein d'une interface. D'autre part, le terme « déclaration » de la définition précédente nous indique que nous nous contenterons de « décrire » les méthodes. Ainsi nous définirons la manière dont doit être construite chacune des méthodes, sans en préciser le fonctionnement interne.
Voici comment procéder à une déclaration d'une méthode :

Code : Actionscript

```
function uneMethode(param:String):void;
```

Identiquement à une déclaration de variable, vous remarquerez qu'on « omet » le contenu de la méthode normalement placé entre accolades. Toutefois, nous retrouvons dans cette déclaration l'ensemble des informations utiles de la méthode, à savoir : le nom de la fonction, les différents paramètres, le type de valeur renvoyé.

Une interface sera donc un ensemble de déclarations de méthodes, regroupées autour d'une utilité commune. Nous pourrons ainsi créer de nouvelles classes qui utilisent ces interfaces et nous redéfinirons l'ensemble des méthodes. On appelle cela l'**implémentation** !

Quel est l'intérêt des interfaces, s'il faut redéfinir l'ensemble des méthodes ?

Étant donné que le contenu des méthodes n'est pas défini à l'intérieur des interfaces, il va bien évidemment falloir le faire dans les classes qui les implémentent. C'est pourquoi il est fort probable que vous vous demandiez à quoi pourrait bien vous servir ce concept. Je vais donc essayer de vous l'expliquer en quelques mots.

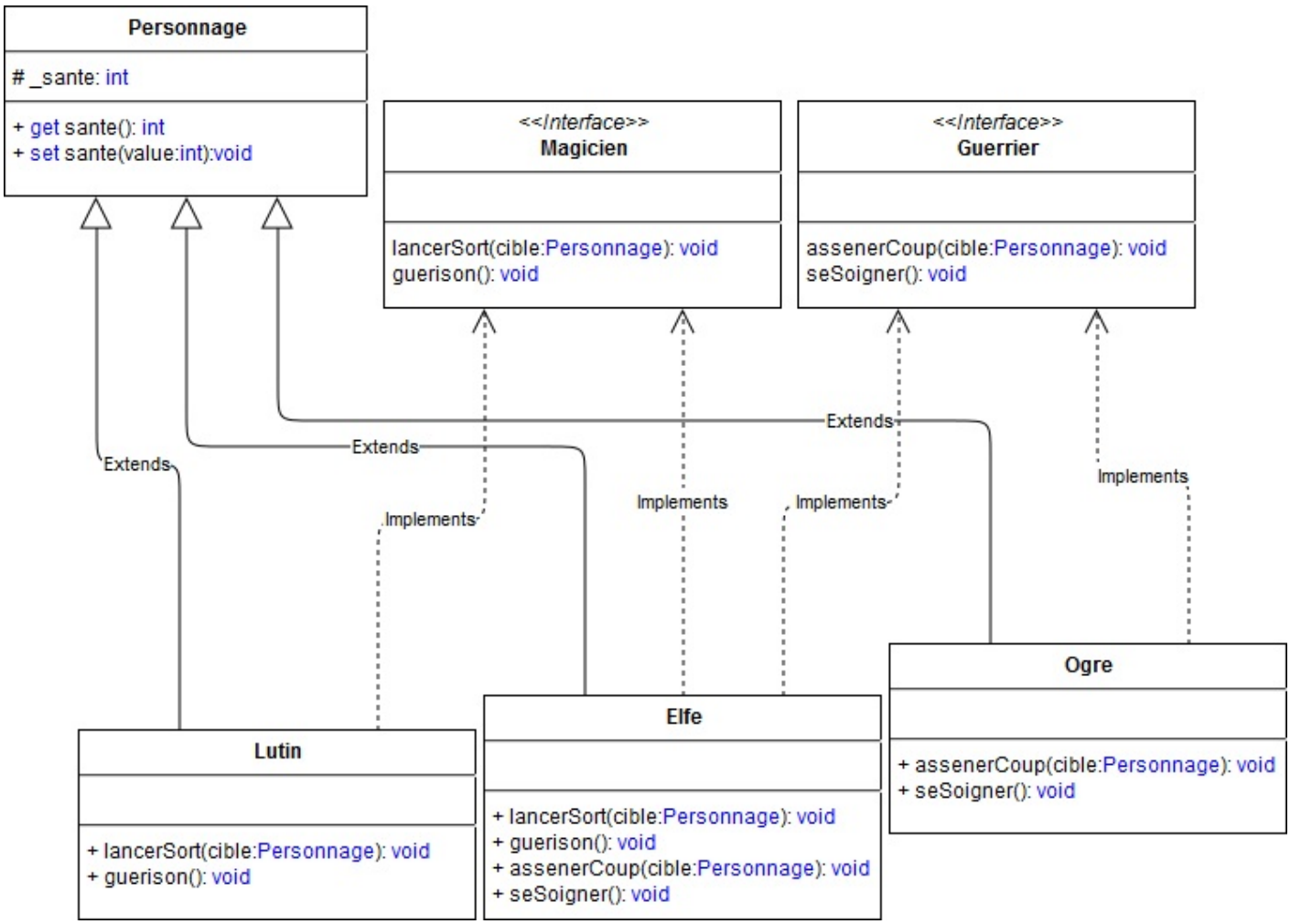
Lorsque nous avions vu la notion d'encapsulation, je vous avais précisé le fait que nous devions masquer le fonctionnement interne d'une classe à l'utilisateur de celle-ci. Ainsi l'utilisation de la classe devenait plus aisée, car nous n'avions plus qu'à manipuler des méthodes explicites qui se chargeaient du travail laborieux.
Une interface va donc nous servir à décrire « l'interface » d'une pseudo-classe, c'est-à-dire présenter celle-ci telle qu'elle est vue de l'extérieur. Nous obtiendrons donc une sorte de « mode d'emploi » qui explique comment cette pseudo-classe doit être manipulée.

Nous conviendrons que le concept d'interfaces n'a d'utilité que dans de gros projets. Cela permet de définir les normes à respecter, et ainsi garantir la compatibilité des classes qui les implémenteront. Ces dernières pourront alors être utilisées de la manière, malgré les différences de contenu à l'intérieur de chacune des méthodes. Cela vient donc compléter la notion de polymorphisme et l'idée de pouvoir utiliser des objets différents comme s'il s'agissait du même.

Ce gros morceau de théorie va maintenant laisser place à la pratique qui, je suis d'accord avec vous, permet en général de mieux cerner la chose.

Les bases de la conception

À présent, nous allons reprendre notre problème de personnages concernant des lutins, des elfes et des ogres !
Parce qu'un schéma est toujours plus explicite que des mots, je vous laisse découvrir à la figure suivante l'ensemble de ces classes et interfaces que nous allons réaliser :



Les classes et les interfaces que nous allons réaliser

En premier lieu, nous allons créer une classe `Personnage` en tant que superclasse pour la suite. Celle-ci n'a rien d'extraordinaire puisqu'elle est identique à ce que nous avons fait précédemment.
Je vous laisse néanmoins le temps de la reprendre avant de passer à la suite :

Code : Actionscript

```
package
{

    public class Personnage
    {

        protected var _sante:int;

        public function Personnage()
        {
            sante = 100;
        }

        public function get sante():int
        {
            return _sante;
        }

        public function set sante(value:int):void
        {
            _sante = value;
        }
    }
}
```

Les interfaces

Nous voilà enfin au cœur du problème, nous allons devoir créer les deux interfaces `Magicien` et `Guerrier` !
Étant donné qu'il s'agit d'un exemple, nous n'allons pas réaliser des dizaines de déclarations de méthodes. Je vous propose ainsi de lier deux méthodes `lancerSort` et `guerison` à notre interface `Magicien`.
Voici donc notre première interface :

Code : Actionscript

```
package
{
    public interface Magicien
    {
        function lancerSort(cible:Personnage):void;
        function guerison():void;
    }
}
```

Déclarons également deux méthodes pour notre seconde interface, nommées `assenerCoup` et `seSoigner`. En voici le résultat :

Code : Actionscript

```
package
{
    public interface Guerrier
    {
        function assenerCoup(cible:Personnage):void;
        function seSoigner():void;
    }
}
```

À présent, nous disposons des bases nécessaires à la création de nos classes « réelles », c'est-à-dire celles que nous utiliserons dans la pratique. Nous pourrons ainsi combiner les notions d'héritage et d'implémentation pour réaliser celles-ci.

Notez que dans cet exemple nous utilisons simultanément les notions d'héritage et d'implémentation. Toutefois en pratique, il n'est pas d'obligatoire de procéder à un héritage pour utiliser les interfaces. Il est donc tout à fait possible d'utiliser uniquement la notion d'implémentation.

L'implémentation

Tout comme nous avions le mot-clé **`extends`** pour l'héritage, nous utiliserons **`implements`** pour implémenter une interface dans une classe. Étant donné que le contenu de chacune des méthodes n'est pas défini dans les interfaces, nous allons devoir le faire ici.

Je vous propose donc de découvrir les trois classes « finales » qui implémenteront donc les interfaces définies juste avant.
Commençons par la classe `Lutin` :

Code : Actionscript

```
package
{

    public class Lutin extends Personnage implements Magicien
    {

        public function Lutin()
        {
            super();
```

```
            }

            public function lancerSort(cible:Personnage):void
            {
                cible.sante -= 10;
                trace("Sort : Boule de feu");
            }
            public function guerison():void
            {
                this.sante += 10;
                trace("Sort : Guérison");
            }
        }
    }
```

La classe `Elfe` est sans doute la plus complexe, puisqu'elle implémente les deux interfaces à la fois :

Code : Actionscript

```
    package
    {

        public class Elfe extends Personnage implements Magicien,
    Guerrier
        {

            public function Elfe()
            {
                super();
            }

            public function lancerSort(cible:Personnage):void
            {
                cible.sante -= 5;
                trace("Sort : Tornade");
            }

            public function guerison():void
            {
                this.sante += 5;
                trace("Sort : Guérison");
            }

            public function assenerCoup(cible:Personnage):void
            {
                cible.sante -= 5;
                trace("Coup : Épée");
            }
            public function seSoigner():void
            {
                this.sante += 5;
                trace("Soin : Herbe médicinale");
            }
        }
    }
```

Lorsqu'on réalise un héritage et une ou plusieurs implémentations en même temps, l'héritage *doit toujours* être effectué en premier. C'est pourquoi le mot-clé **`extends`** sera toujours placé *avant* **`implements`** dans la déclaration de la classe. Vous remarquerez également que les différentes interfaces implémentées ici sont séparées par une virgule.

Enfin pour finir, la classe `Ogre` ne devrait maintenant plus poser de problèmes :

Code : Actionscript

```
package
{

    public class Ogre extends Personnage implements Guerrier
    {

        public function Ogre()
        {
            super();
        }

        public function assenerCoup(cible:Personnage):void
        {
            cible.sante -= 10;
            trace("Coup : Hache de guerre");
        }
        public function seSoigner():void
        {
            this.sante += 10;
            trace("Soin : Bandage");
        }
    }
}
```

Sachant que les méthodes ne sont que déclarées à l'intérieur des interfaces, celles-ci doivent *obligatoirement* être redéfinies dans les classes qui les implémentent. Veillez donc à bien toutes les redéfinir sous peine de messages d'erreurs lors de la compilation de votre projet.

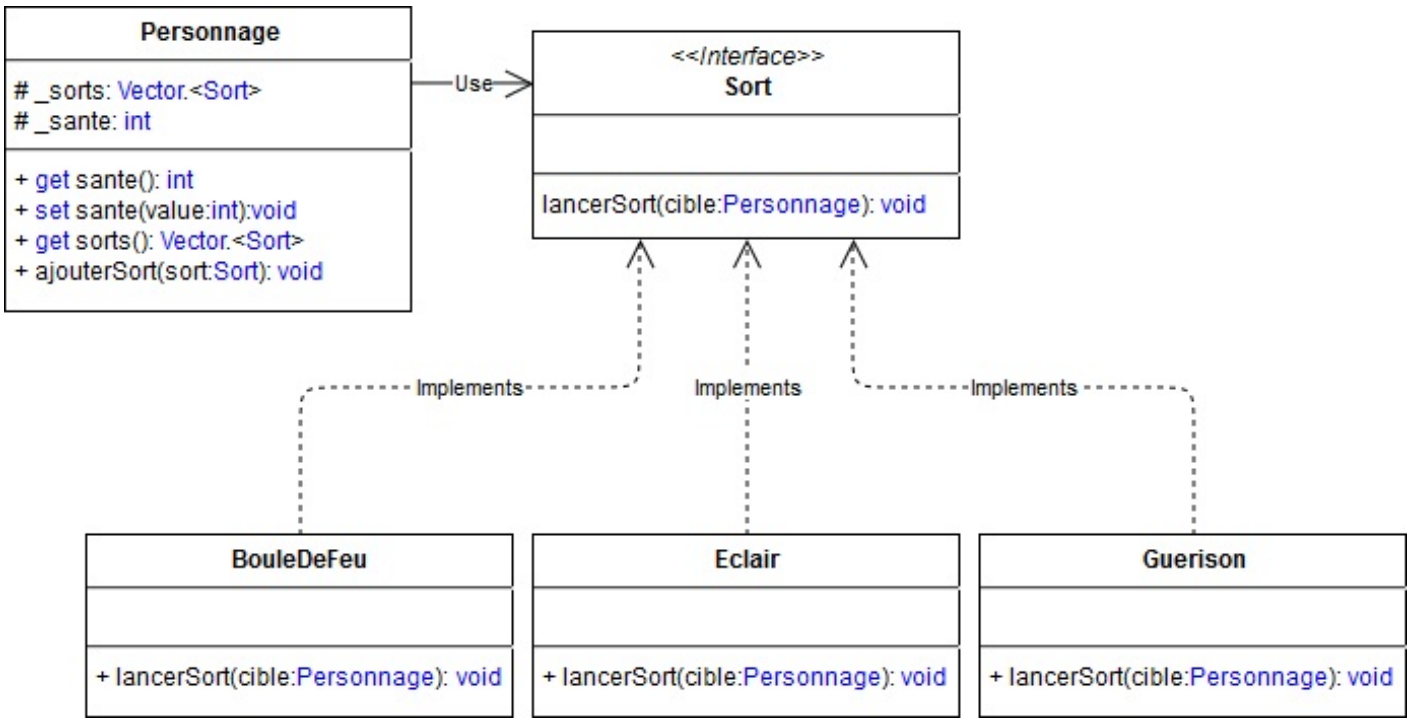
Plus loin avec les interfaces

Nous avons vu le principe des interfaces, mais je vais ici vous présenter une utilisation qui en fait tout leur intérêt !

Imaginez maintenant que vous vouliez créer un jeu beaucoup plus complet, qui risque d'évoluer dans une version ultérieure. Vous voulez cette fois multiplier le nombre des races des personnages avec, par exemple, les classes `Humain`, `Nain`, `Lutin`, `Elfe`, `Centaure`, `Minotaure`, `Ogre`, `Gnome` et `Troll`. Cette fois vous n'avez plus de simples magiciens ou de purs guerriers, mais chacune des races dispose de l'ensemble des facultés. Ainsi chaque espèce est capable de lancer des sorts ou de se battre avec n'importe quel type d'armes. En revanche, chacun possède une liste limitée de sorts au départ, mais peut la développer au fil du jeu. Tous ces sorts sont utilisés de la manière, seuls leurs effets et leurs dégâts sont différents de l'un à l'autre.

Une architecture judicieuse de vos classes serait alors de créer une interface `Sort`, qui serait implémentée par chacun de vos divers sorts. L'intérêt ici serait alors de pouvoir créer une liste de sort de type `Vector.<Sort>` comme attribut de votre classe `Personnage`. Chaque sort serait alors utilisé de la même façon quel que soit celui-ci.

Voici un schéma à la figure suivante qui résume la situation et qui devrait plus facilement vous séduire.



Créez une interface « Sort »

Vous pouvez ainsi créer un nouvel `Ogre` nommé `budoc`. Héritant de la classe `Personnage`, votre personnage va pouvoir apprendre de nouveaux sorts au fil du jeu. Voici par exemple notre ami `budoc` qui apprend les sorts `Guerison` et `Eclair` :

Code : Actionscript

```
var budoc:Ogre = new Ogre();
budoc.ajouterSort(new Guerison());
budoc.ajouterSort(new Eclair());
```

Je vous ai présenté le concept avec les interfaces, mais sachez que cette pratique peut également être utilisée avec la relation d'héritage, même si cela est moins courant. Au risque de me répéter encore une fois, une des grandes forces de la POO est justement de pouvoir se servir d'objets différents de la même manière grâce à ces techniques d'héritage, de polymorphisme et d'implémentation.

Nous sommes maintenant arrivés au bout de ces explications sur les interfaces. J'espère vous avoir convaincus de leur utilité et j'ose croire que vous les utiliseraient de la bonne façon.

Les classes abstraites

Le concept

Une classe abstraite est une classe que l'on ne peut pas instancier, c'est-à-dire que l'on ne peut pas directement créer d'objet de cette classe avec le mot-clé **new**. Lorsque l'on choisit de rendre une classe abstraite, on restreint ainsi ses possibilités d'utilisation. Par contre, les sous-classes de cette classe abstraite ne sont pas forcément abstraites, à moins de l'indiquer à chaque fois !

Pour comprendre pourquoi cela pourrait améliorer la cohérence de votre code, reprenons l'exemple des véhicules sur l'autoroute (voir figure suivante).

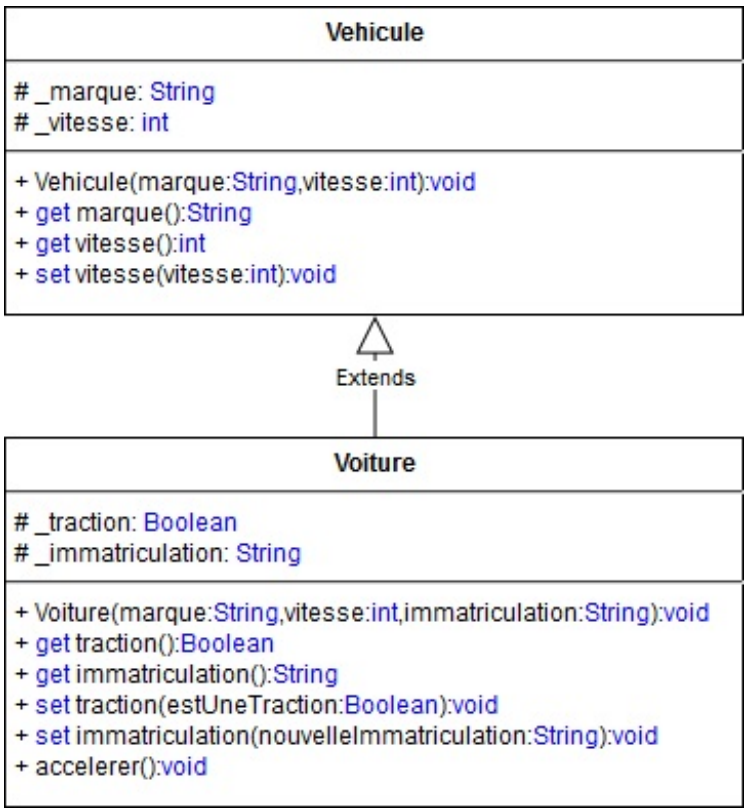


Diagramme UML des classes Vehicule et Voiture

Il était alors possible de créer directement un véhicule :

Code : Actionscript

```
var vehicule:Vehicule = new Vehicule('Renault', 0);
```

Maintenant, si notre application ne s'occupe que de voitures, il serait inutile d'utiliser directement la classe `Vehicule` alors que nous disposons de sa sous-classe `Voiture`. Si nous laissions à d'autres programmeurs la possibilité de créer des véhicules au lieu de voitures, ils auraient une chance sur deux de se tromper et d'oublier d'utiliser la classe `Voiture`. Pour cela, il faut rendre la classe `Vehicule` abstraite (voir figure suivante) !

Par opposition à une classe abstraite, une classe qui peut être instanciée est une classe **concrète**.

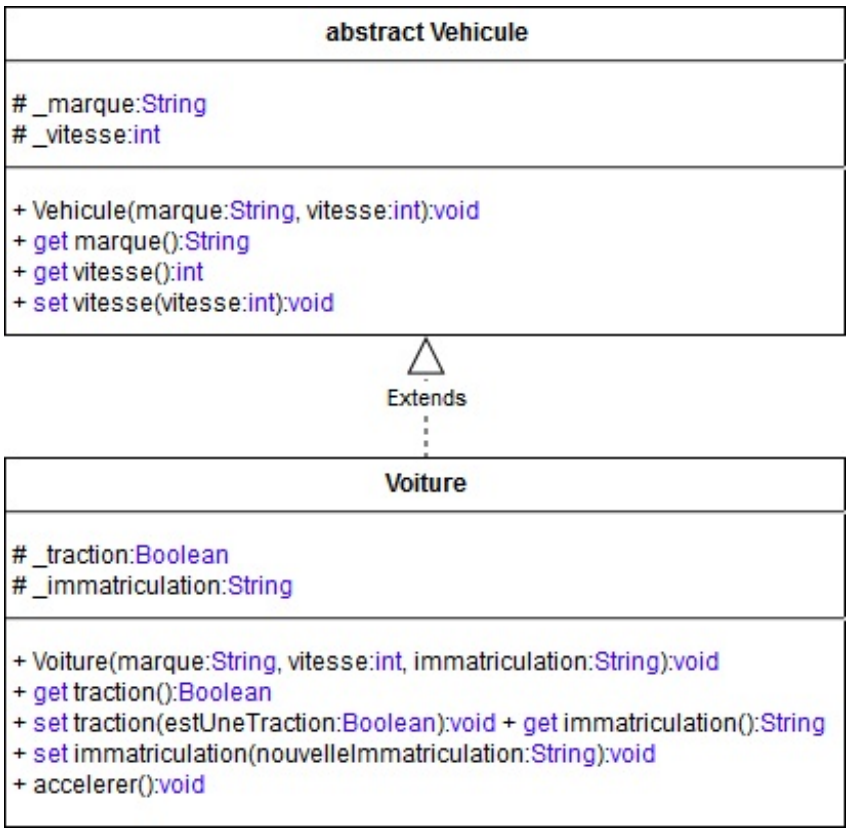


Diagramme UML de la classe abstraite Vehicule et de sa sous-classe

Rappel : une sous-classe d'une classe abstraite n'est pas forcément abstraite ; il faut le spécifier à chaque fois. Dans notre exemple, on peut donc instancier la classe `Voiture`, mais pas la classe `Vehicule`.

Ainsi, nous sommes désormais obligés d'utiliser la classe `Voiture` pour créer un objet voiture : notre code est **cohérent** ! 

Code : Actionscript

```
var vehicule:Vehicule = new Vehicule('Renault', 0); // Interdit !

var voiture:Voiture = new Voiture('Renault', 0, 'BX6F57'); //
Autorisé :)
```

Application à l'ActionScript 3

Malheureusement, ce principe de classes abstraites ne dispose pas de mécanismes officiels au sein du langage ; il faut donc le « programmer » soi-même. Cela ne signifie pas qu'il ne faut pas l'utiliser, et plusieurs classes fournies par *Flash* sont des classes abstraites ! Mais rassurez-vous, c'est en réalité très simple. 

Le principe est le suivant : nous allons faire une vérification dans le constructeur de la classe-mère abstraite, à l'aide d'un paramètre et du mot-clé **`this`**. En effet, seule une sous-classe de cette classe peut passer en paramètre du constructeur le mot-clé **`this`** !

Voici le code de la classe abstraite :

Code : Actionscript

```
package
{
    import flash.errors.IllegalOperationError;
    /**
```

```
/*
* Une classe abstraite
*/
    public class MaClasseAbstraite
    {

        public function MaClasseAbstraite(moi:MaClasseAbstraite)
        {
            // Nous vérifions si le paramètre de vérification
correspond à l'objet
            if (moi != this)
            {
                // Sinon, quelqu'un tente d'instancier cette classe,
nous envoyons donc une erreur
                // car seule une sous-classe peut passer le mot-clé
'this' en paramètre au constructeur
                throw new IllegalOperationError("MaClasseAbstraite
est une classe abstraite et ne peut donc pas être directement
instanciée.");
            }
        }

    }

}
```

Le mot-clé **throw** permet d'envoyer une erreur ; ici `IllegalOperationError`, qui indique qu'il s'agit d'une opération illégale. Pour l'instant il n'est pas nécessaire que vous vous attardiez sur cette instruction, puisque nous reviendrons sur la gestion des erreurs dans un chapitre spécialisé, plus loin de ce cours.

Nous demandons dans le constructeur un paramètre `moi` obligatoire, du même type que la classe elle-même : ainsi, seuls les objets des sous-classes de `MaClasseAbstraite` seront acceptés. Ensuite, nous vérifions que ce paramètre *pointe* vers le même objet, c'est-à-dire que cet objet qui vient d'être créé est bien une instance d'une sous-classe. 

Rappel : le mot-clé **this** est une référence qui pointe sur l'unique objet qui fait actuellement travailler le corps de la classe. Même en remontant entre les sous-classes, cette référence pointe toujours sur le même objet (qui appartient à toutes ces classes en même temps, comme nous l'avons vu dans le chapitre sur l'héritage).

Voici le code de la sous-classe concrète :

Code : Actionscript

```
package
{
    /**
* Une sous-classe concrète.
*/
    public class MaClasseConcrete extends MaClasseAbstraite
    {

        public function MaClasseConcrete()
        {
            // On envoie la référence de l'objet qui vient d'être
créé à la classe mère pour passer la vérification
            super(this);

        }

    }

}
```

Dans le constructeur de la classe-fille concrète, nous appelons le constructeur de la classe-mère abstraite à l'aide du mot-clé

super pour passer la vérification.

Voici ce que cela donne lorsque nous voulons créer un objet de chaque classe :

Code : Actionscript

```
var objet1:MaClasseAbstraite = new MaClasseAbstraite(); // Une
erreur va être envoyée

var objet2:MaClasseConcrete = new MaClasseConcrete(); // Tout va
bien.
```

Les types inconnus

Déterminer si un objet est une occurrence d'une certaine classe

Le mot-clé `is` permet de renvoyer un booléen pour savoir si un objet quelconque est une occurrence d'une certaine classe, ou d'une sous-classe.

Code : Actionscript

```
var entier:int = -23;
var entierPositif:uint = 42;
var nombre:Number = 3.14;
var chaine:String = 'Hello world!';

// entier est du type int
trace(entier is int); // Affiche: true
// entier n'est pas du type uint
trace(entier is uint); // Affiche: false
// La classe int est une sous-classe de Number
trace(entier is Number); // Affiche: true

trace(entierPositif is uint); // Affiche: true
// La classe uint est aussi une sous-classe de Number
trace(entierPositif is Number); // Affiche: true

trace(nombre is int); // Affiche: false
trace(nombre is Number); // Affiche: true

trace(chaine is String); // Affiche: true
trace(chaine is int); // Affiche: false
```

Cela fonctionne également avec les fonctions ! Car il faut toujours se rappeler qu'en ActionScript 3, tout est objet : les variables, les fonctions, les objets de la classe `Voiture`, leurs attributs, leurs méthodes...

Code : Actionscript

```
function maFonction():void
{
    trace('Je suis dans une fonction !');
}

// Toute fonction est une instance de la classe Function !
trace(maFonction is Function); // Affiche: true
// Mais chaine n'est pas une fonction :p
trace(chaine is Function); // Affiche: false

// Tout objet est une instance de la classe Object !
trace(entier is Object); // Affiche: true
trace(chaine is Object); // Affiche: true
trace(maFonction is Object); // Affiche: true
```

Des paramètres de type inconnu

Dans les fonctions, il est possible de demander des paramètres de type inconnu, c'est-à-dire dont nous ne pouvons pas connaître la classe à l'avance ! Pour cela, nous pouvons utiliser la classe `Object`, qui est la classe mère de tous les objets en ActionScript 3.

Code : Actionscript

```
function quiSuisJe(quelqueChose:Object):void
{
    if(quelqueChose is Function)
    {
        trace('Je suis une fonction !');
    }
}
```

Il existe un raccourci pour spécifier que le type d'un paramètre est inconnu : le caractère « * » ! Reprenons l'exemple précédent en utilisant ce raccourci :

Code : Actionscript

```
function quiSuisJe(quelqueChose:*):void
{
    if(quelqueChose is Function)
    {
        trace('Je suis une fonction !');
    }
}
```

Essayons cette fonction avec une autre fonction :

Code : Actionscript

```
function maFonction():void
{
    trace('Je suis dans une fonction !');
}

function quiSuisJe(quelqueChose:*):void
{
    if(quelqueChose is Function)
    {
        trace('Je suis une fonction !');
    }
}

// Nous passons l'objet maFonction de classe Function en paramètre
!
quiSuisJe(maFonction); // Affiche: Je suis une fonction !
```

L'objet `maFonction` est une occurrence de la classe `Function` (étant donné que c'est une fonction), donc le test est positif et le message « Je suis une fonction ! » est affiché.

Nous pourrions ensuite appeler la fonction à l'aide de la méthode `call()` de la classe `Function` :

Code : Actionscript

```
function maFonction():void
{
    trace('Je suis dans une fonction !');
```

```
}

function quiSuisJe(quelqueChose:*):void
{
    if(quelqueChose is Function)
    {
        trace('Je suis une fonction !');
        // On appelle la fonction !
        quelqueChose.call();
    }
}

// Nous passons l'objet maFonction de classe Function en paramètre
quiSuisJe(maFonction); // Affiche: Je suis une fonction ! Je suis
dans une fonction !
```

Accéder dynamiquement aux propriétés

En ActionScript 3, il est possible d'accéder à une propriété d'un objet en connaissant son nom uniquement à l'exécution. Par exemple, vous obtenez le nom d'un attribut dans une variable de type `String`, et vous voulez ensuite modifier sa valeur pour faire une animation ou autre chose.

Prenons une classe `Voiture` disposant d'un attribut `vitesse` et d'une méthode `accelerer` :

Code : Actionscript

```
package
{
    public class Voiture
    {

        private var _vitesse:int;

        public function Voiture()
        {
            vitesse = 10;
        }

        public function get vitesse():int
        {
            return _vitesse;
        }

        public function set vitesse(value:int):void
        {
            _vitesse = value;
        }

        public function accelerer(combien:int = 1):void
        {
            vitesse += combien;
        }
    }

}
```

La syntaxe à respecter pour accéder à l'attribut de nom `'vitesse'` est la suivante :

Code : Actionscript

```
var objet:Voiture = new Voiture();
var nomDeLaPropriete:String = 'vitesse';

trace(objet[nomDeLaPropriete]); // Affiche la vitesse de la
voiture: 10
```

Ou plus simplement :

Code : Actionscript

```
var objet:Voiture = new Voiture();
trace(objet['vitesse']); // Affiche: 10
```

Rappel : une propriété constituée d'un ou plusieurs accesseurs est considérée comme un attribut.

En guise d'exemple, supposons que nous voulons disposer d'une fonction qui augmente de 10 n'importe quel attribut numérique d'un objet quelconque, sans avoir besoin de plus de précisions concernant sa classe.

Code : Actionscript

```
function augmenter(objet:Object, propriete:String):void
{
    // On vérifie le type de la propriété : il faut que ce soit un
nombre
    if(objet[propriete] is int || objet[propriete] is uint ||
objet[propriete] is Number)
    {
        objet[propriete] += 10;
    }
    else
    {
        trace("Attention : La propriété " + propriete + " sur
l'objet " + objet + " n'est pas un nombre.");
    }
}
```

Écrivons un petit programme principal qui crée un objet `Voiture` et qui utilise cette fonction pour augmenter sa vitesse :

Code : Actionscript

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;

    /**
* Programme principal
*/
    public class Main extends Sprite
    {

        public function Main():void
        {
            if (stage)
                init();
            else
                addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void
        {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // entry point

            // On créé une voiture
```

```
                var voiture:Voiture = new Voiture();
                // On initialise sa vitesse à 10
                voiture.vitesse = 10;
                // On augmente sa vitesse de 10
                augmenter(voiture, 'vitesse');
                trace(voiture.vitesse); // Affiche: 20

            }

            public function augmenter(objet:Object,
    propriete:String):void
            {
                // On vérifie le type de la propriété : il faut que ce
    soit un nombre
                if (objet[propriete] is int || objet[propriete] is uint
    || objet[propriete] is Number)
                {
                    objet[propriete] += 10;
                }
                else
                {
                    trace("Attention : La propriété " + propriete + "
    sur l'objet " + objet + " n'est pas un nombre.");
                }
            }

        }

    }
```

Est-ce que cela fonctionne pour les méthodes ?

La réponse est oui, nous pouvons aussi le faire pour les méthodes ! Il suffit d'ajouter des paramètres comme pour n'importe quel appel de fonction. Par exemple :

Code : Actionscript

```
var voiture:Voiture = new Voiture();
voiture.vitesse = 10;

// Appelons la méthode 'accelerer' de notre objet voiture !
voiture['accelerer'](5);
trace(voiture.vitesse); // Affiche: 15

// Ce qui revient à écrire :
voiture.accelerer(5);
trace(voiture.vitesse); // Affiche: 20

// Avec les accesseurs, la propriété vitesse est concidérée comme
un attribut
trace(voiture['vitesse'] is Function); // Affiche: false
// Alors que accelerer est bien une méthode
trace(voiture['accelerer'] is Function); // Affiche: true
```

Un exemple plus concret ; nous pourrions coder une fonction qui anime une propriété numérique d'un objet d'une valeur à une autre, pendant un certain temps, mais malheureusement, nous ne sommes pas encore assez loin dans le cours pour pouvoir faire cela. Toutefois, je vous montre à quoi pourrait ressembler la signature de la fonction et un exemple d'utilisation :

Code : Actionscript

```
function animer(objet:Object, propriete:String, valeurDepart:*,
valeurArrivee:*, duree:Number):void
{
```

```
        // Animation à l'aide de objet[propriete]
    }
```

J'utilise la notation avec l'étoile pour le type de `valeurDepart` et `valeurArrivee` pour que l'on puisse passer des nombres de type `int`, `uint` ou `Number`. Attention toutefois à penser à vérifier leurs types dans le corps de la fonction.

Dans notre classe `Main`, nous pourrions écrire :

Code : Actionscript

```
var voiture:Voiture = new Voiture();

// Accélérons !
animer(voiture, 'vitesse', 0, 100, 3); // Nous animons la vitesse de
l'objet voiture de 0 à 100 pendant 3 secondes
```

Avec cette fonction, nous pourrions donc animer n'importe quel attribut de n'importe quel objet ! 

- Il est possible de créer des classes dynamiques que l'on peut modifier durant l'exécution, grâce au type **`dynamic`**.
- Les interfaces permettent d'hériter en quelque sorte de plusieurs classes, pour partager des propriétés communes.
- Nos classes peuvent être abstraites : dans ce cas, on renvoie une erreur si quelqu'un essaye d'en créer une instance.
- Le type des variables que nous manipulons peut être indéfini ; on peut alors le tester à l'aide du mot-clé `is`.
- Il est possible d'accéder dynamiquement à des propriétés dont on connaît le nom.

Partie 3 : L'affichage

Les objets d'affichage

Dans ce chapitre, nous allons aborder le système qui va nous permettre (enfin !) d'afficher des éléments (texte, dessins, images, etc.) à l'écran ! Mais nous ne pourrons réellement afficher quelque chose qu'au chapitre suivant, car il nous faut aborder quelques notions de base auparavant.

Nous allons voir notamment :

- comment décrire une couleur en ActionScript,
- le fonctionnement l'affichage sur un écran,
- et enfin le système d'affichage de Flash.

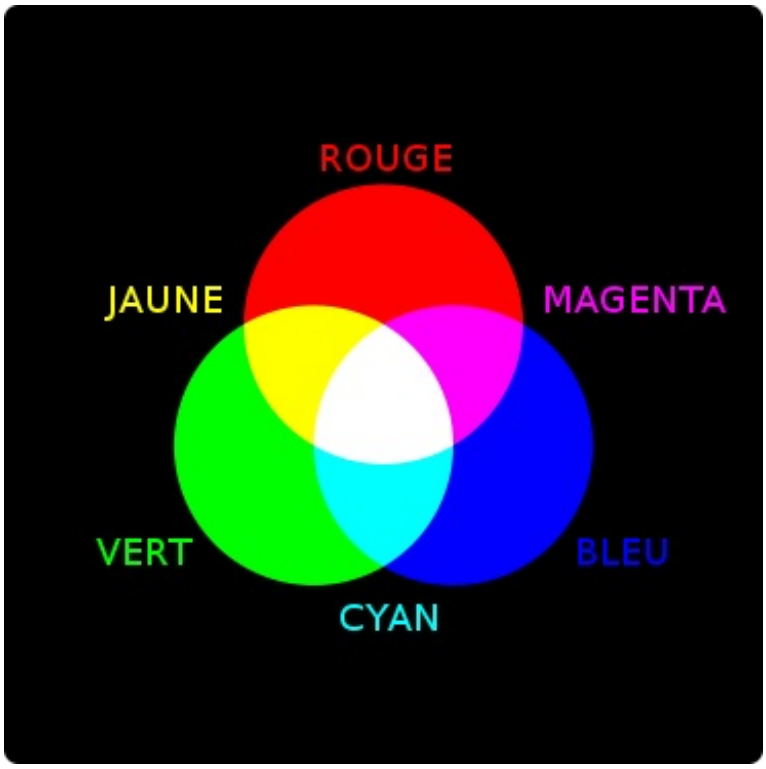
C'est parti !

Introduction

Les couleurs

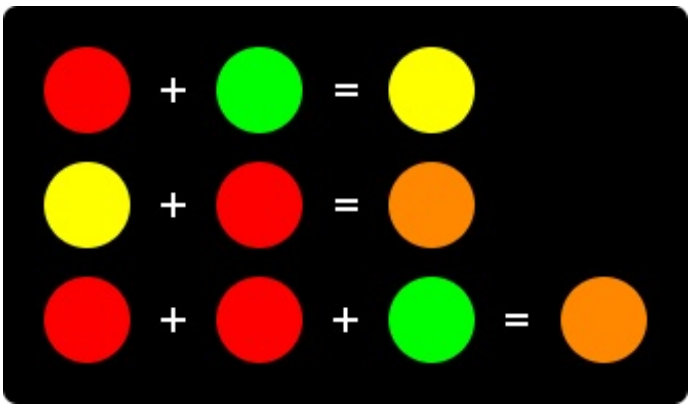
Composer une couleur

En informatique, les couleurs sont décomposées en couleurs principales : le rouge, le vert et le bleu. À partir de ces trois couleurs, il est possible de créer n'importe quelle autre couleur en les dosant (voir figure suivante).



Les couleurs principales de la lumière

Exemple : je souhaite créer une couleur orange. L'orange est constitué de jaune et de rouge. Ainsi, il nous faut d'abord faire du jaune (rouge + vert), puis rajouter du rouge (voir figure suivante). Au final, l'orange se crée grosso-modo avec deux quantités de rouge pour une quantité de vert.



Composition de la couleur orange

Notation hexadécimale

Pour décrire une couleur composée de rouge, de vert et de bleu, nous utiliserons la **notation hexadécimale**. On commence par écrire `0x` pour signifier que l'on utilise la notation hexadécimale, puis on décrit les quantités de chacune des trois couleurs rouge, vert et bleu : deux chiffres, allant de 0 (le minimum) à F (le maximum) pour chacune, ce qui fait six chiffres.

Pourquoi jusqu'à F ? Depuis quand F est un chiffre ?

Excellente question ! En réalité, il existe différents systèmes de notation pour les nombres. Celle que vous et moi utilisons tous les jours s'appelle la **notation décimale**. Cela signifie que nous utilisons les chiffres de 0 à 9 pour écrire les nombres. On dit aussi que nous écrivons les nombres en **base dix**.
Une autre notation très connue est la **notation binaire** : nous décrivons les nombres avec seulement les chiffres 0 et 1. On parle alors de **base deux**. Par exemple, `5` en base dix s'écrit `101` en base deux !

La notation hexadécimale utilise non pas dix, mais seize chiffres ! On parle de **base seize** pour cette notation. Les chiffres vont de 0 à 15 ; avouez cependant qu'écrire un chiffre à l'aide de deux chiffres serait plutôt embêtant... Comment faire alors la différence entre 12 (le chiffre) et 12 (le nombre composé du chiffre 1 et du chiffre 2) ? Impossible !

Ainsi, nous utilisons les 6 premières lettres de l'alphabet pour remplacer les chiffres 10 à 15, ce qui est tout de même bien plus pratique.

Chiffre (décimal)	Notation hexadécimale
10	A
11	B
12	C
13	D
14	E
15	F

En dehors de l'ActionScript, on utilise plus généralement le # pour commencer une notation hexadécimale.

Ainsi, pour écrire notre chiffre 12, nous utilisons C ! Voici quelques exemples de nombre en base dix convertis en base seize :

Base dix	Base seize
12	C
5	5
31	1F

32	20
16	10
160	A0
66	42

Afin de décrire une couleur, nous utiliserons donc deux chiffres en hexadécimal pour le rouge, le vert et le bleu, de `00` à `FF`, ce qui fait de `0` à `255` en décimal. Par exemple, pour notre couleur orange, nous écrirons : `0xFF8000`, car il y a deux doses de rouge pour une dose de vert (voir figure suivante).

Notation hexadécimale de la couleur orange

La plupart des logiciels de dessin et retouche d'image affichent la notation hexadécimale des couleurs, qu'il est alors facile de copier-coller. Il existe également des outils ou des sites web spécialisés dans la création de couleur, qui fournissent cette notation (comme par exemple, ColRD). *Attention toutefois, il ne faut pas oublier qu'en ActionScript, la notation hexadécimale commence par `0x` et non pas par # !*

Stocker une couleur dans une variable

Le type qu'il faut utiliser pour stocker un nombre en notation hexadécimale est le type `uint` :

Code : Actionscript

```
var orange:uint = 0xFF8000;
var orange2:uint = 0xff8000; // Les lettres peuvent aussi être notées en minuscule
```

La variable ainsi créée peut être manipulée comme n'importe quelle autre variable :

Code : Actionscript

```
trace("La couleur orange vaut " + orange + " en base dix."); // 16744448
trace("La couleur orange vaut " + orange.toString(10) + " en base dix."); // 16744448
trace("La couleur orange vaut " + orange.toString(16) + " en base seize."); // FF8000
trace("La couleur orange vaut " + orange.toString(2) + " en base deux."); // 1111 1111 1000 0000 0000 0000
```

Vous remarquerez que la couleur est stockée sous la notation décimale, car c'est la seule notation supportée par les types de nombres. Par contre, il est facile de retrouver la notation hexadécimale ou binaire grâce à la méthode `toString()` comme dans l'exemple ci-dessus.

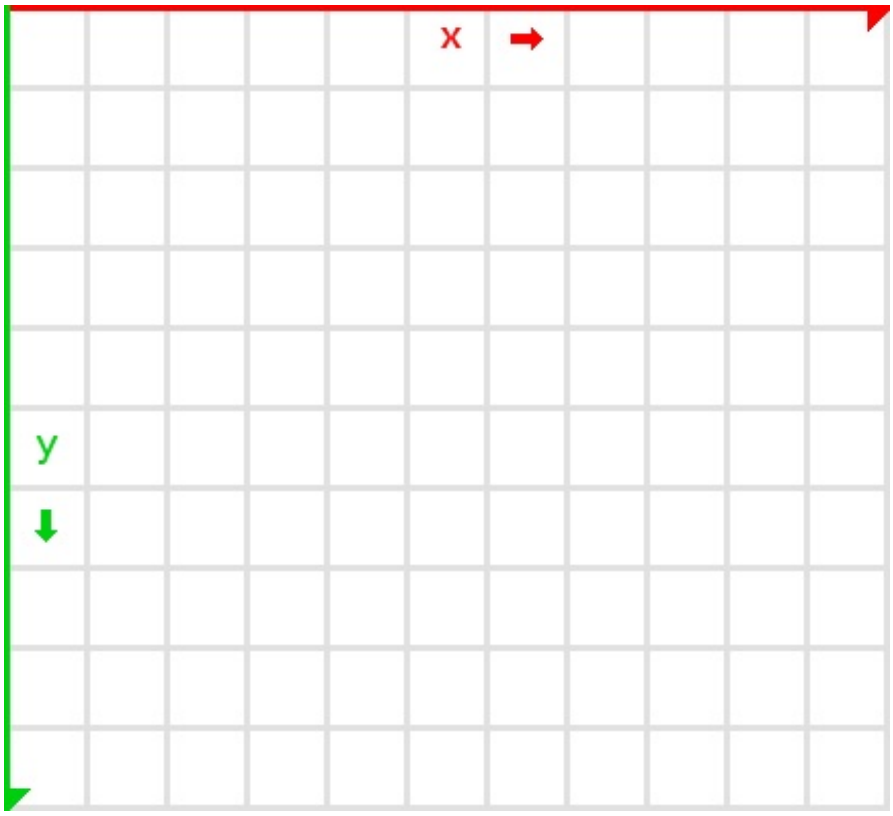
L'affichage sur un écran

Tous les écrans d'affichage sont composés d'une grille de petits carrés de lumière que l'on nomme **pixels** (voir figure suivante). Nous allons voir comment utiliser cette grille pour positionner des objets.

L'écran est composé d'une grille de pixels

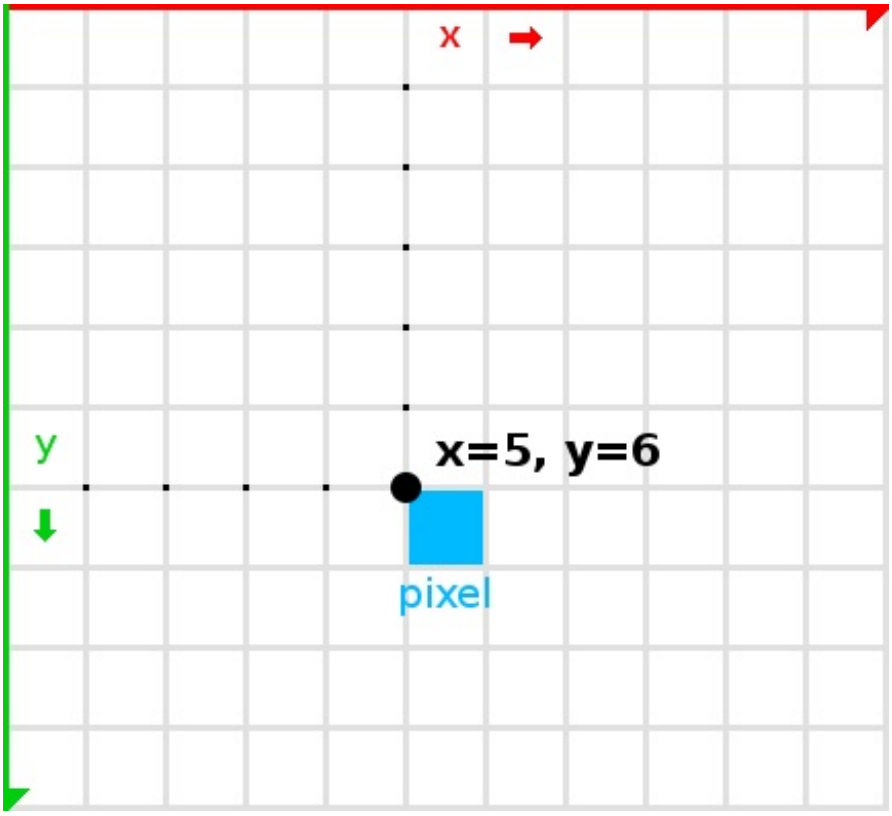
Le repère

Les pixels forment donc une grille suivant deux axes : l'axe des X et l'axe des Y. On se situe alors sur un espace en 2D. Pour mieux visualiser la grille, nous allons utiliser à la figure suivante un repère, qui nous montre les axes et l'origine de la grille.



Repère de l'affichage

Une position (c'est-à-dire un point précis de ce repère) est décrite par deux coordonnées : la position en X, et la position en Y. On écrit toujours les coordonnées en 2D dans cet ordre : d'abord les X, puis les Y. Ainsi il n'est pas nécessaire de tout le temps préciser qui est quoi. Dans l'exemple à la figure suivante, le pixel que l'on a colorié en bleu se trouve à la position (5,6), c'est-à-dire à 5 pixels du bord gauche de l'écran, et à 6 pixels du bord supérieur de l'écran.

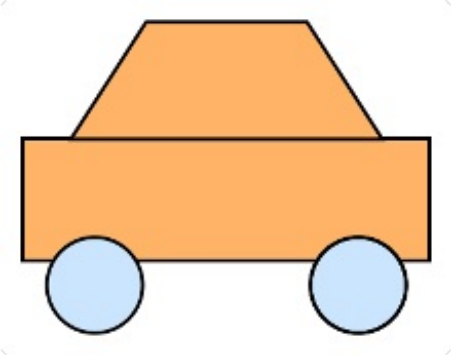


Exemple de position sur l'écran

Tout au long du cours, il faudra avoir ce repère en tête dès qu'il s'agira de positionner un objet à l'écran. Il faut savoir qu'il est possible que les coordonnées soient négatives, mais dans ce cas, les pixels qui dépassent ne seront pas affichés (faute de pixels sur votre écran).

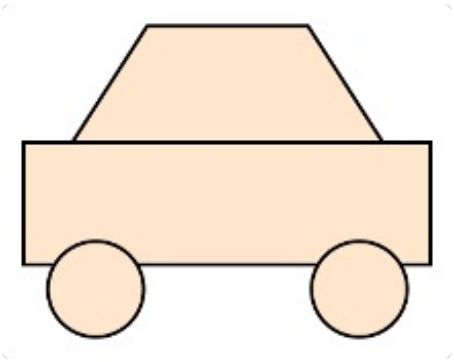
L'arbre des objets d'affichage
L'arbre d'affichage

Pour bien comprendre de quoi il s'agit, commençons par un exemple concret, et revenons sur notre chère voiture. Nous voulons maintenant l'afficher et l'animer sur notre écran ! Nous allons utiliser un modèle simplifié, celui de la figure suivante.



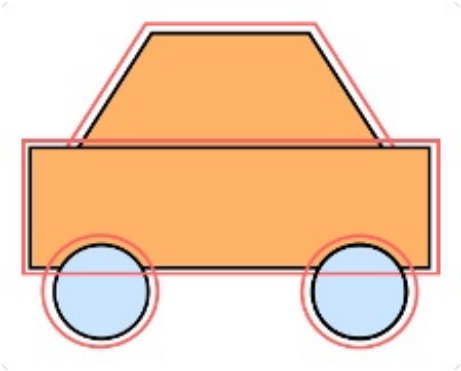
Votre magnifique voiture simplifiée

Supposons que cette voiture soit composée d'une carrosserie et de deux roues. Vue de l'extérieur, la voiture est un objet et un seul (voir figure suivante).



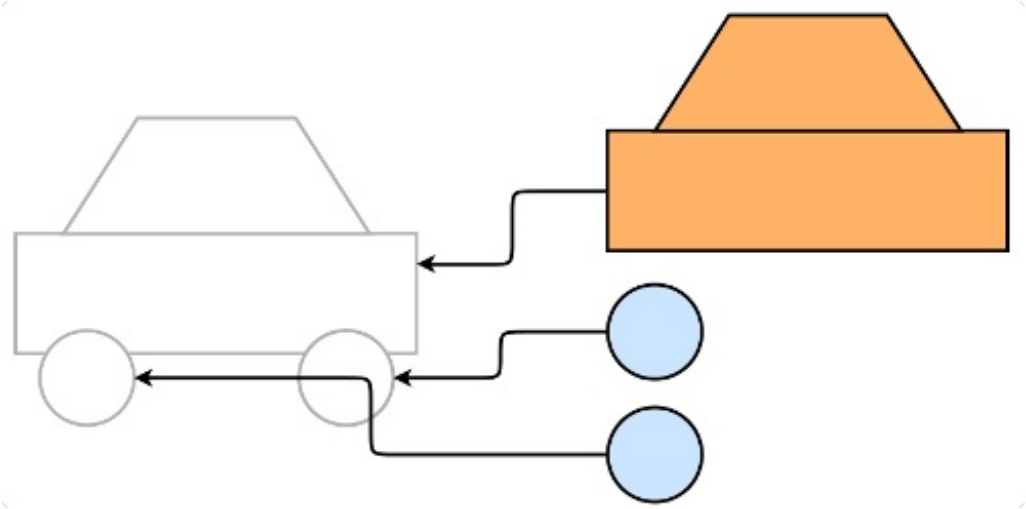
La voiture vue de l'extérieur

Mais en réalité, il y a trois objets *à l'intérieur* de cet objet (voir figure suivante) : la carrosserie en orange et les deux roues en bleu.



L'objet Voiture est composé d'autres objets

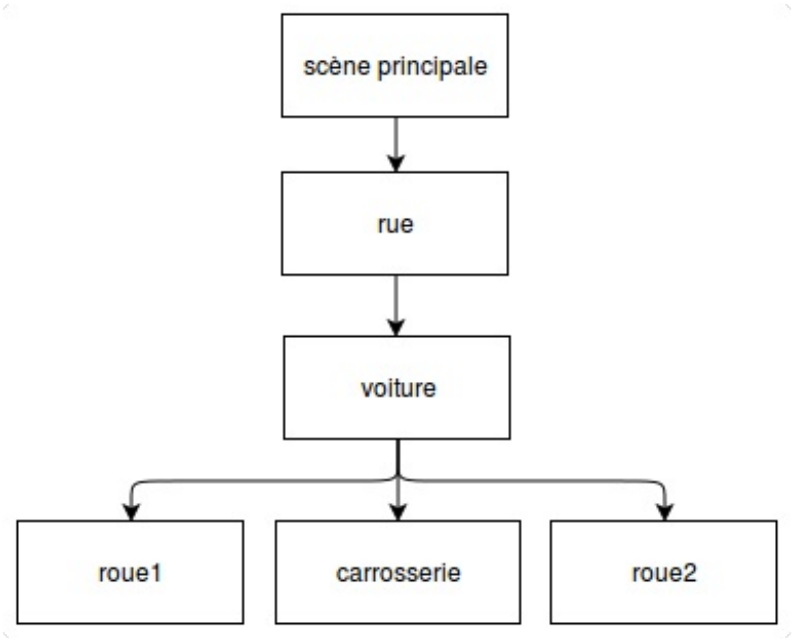
Lorsque nous voudrons afficher la voiture, nous créerons tout d'abord un **conteneur** (par exemple, avec la classe `Sprite`), puis nous ajouterons à l'intérieur trois autres objets d'affichage (voir figure suivante) : un pour la carrosserie, et un pour chaque roue.



Nous disposons les différents objets dans le conteneur Voiture

Ainsi, si nous bougeons le conteneur « voiture », les roues se déplaceront en même temps que la carrosserie ; si nous tournons le conteneur, l'ensemble fera une rotation comme s'il s'agissait d'un unique objet, à l'instar de ce qui se passerait dans la vraie vie ! La voiture pourrait elle-même se trouver dans un autre conteneur "rue", que l'on pourrait faire défiler comme un tout…

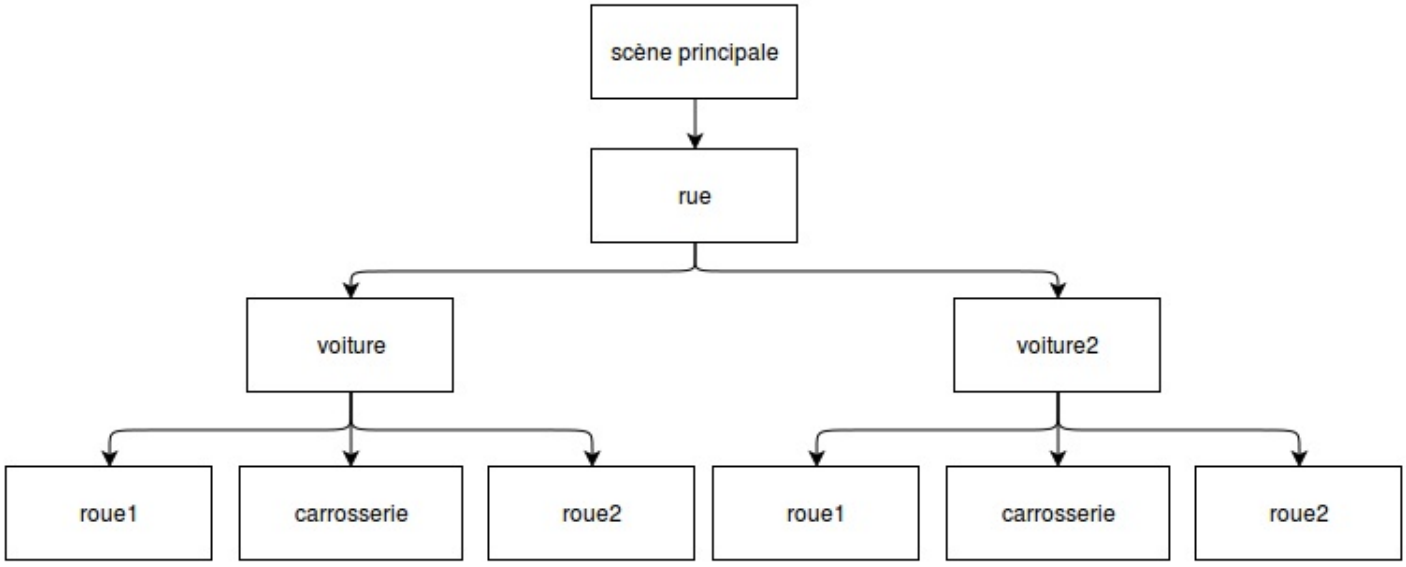
Ainsi, l'arbre d'affichage est une arborescence d'objets d'affichage inclus les uns dans les autres dans une certaine logique. Voici un schéma à la figure suivante représentant l'arbre d'affichage de notre application



L'arbre d'affichage de la voiture

Vous remarquerez que la racine de l'arbre est composée d'un objet que j'ai appelé *scène principale*. C'est un objet spécial de la classe `Stage` qui est toujours présent pour la gestion de l'affichage.

Maintenant, si nous voulons qu'il y ait deux voitures dans la rue, il suffit d'en ajouter une autre dans le conteneur « rue ».

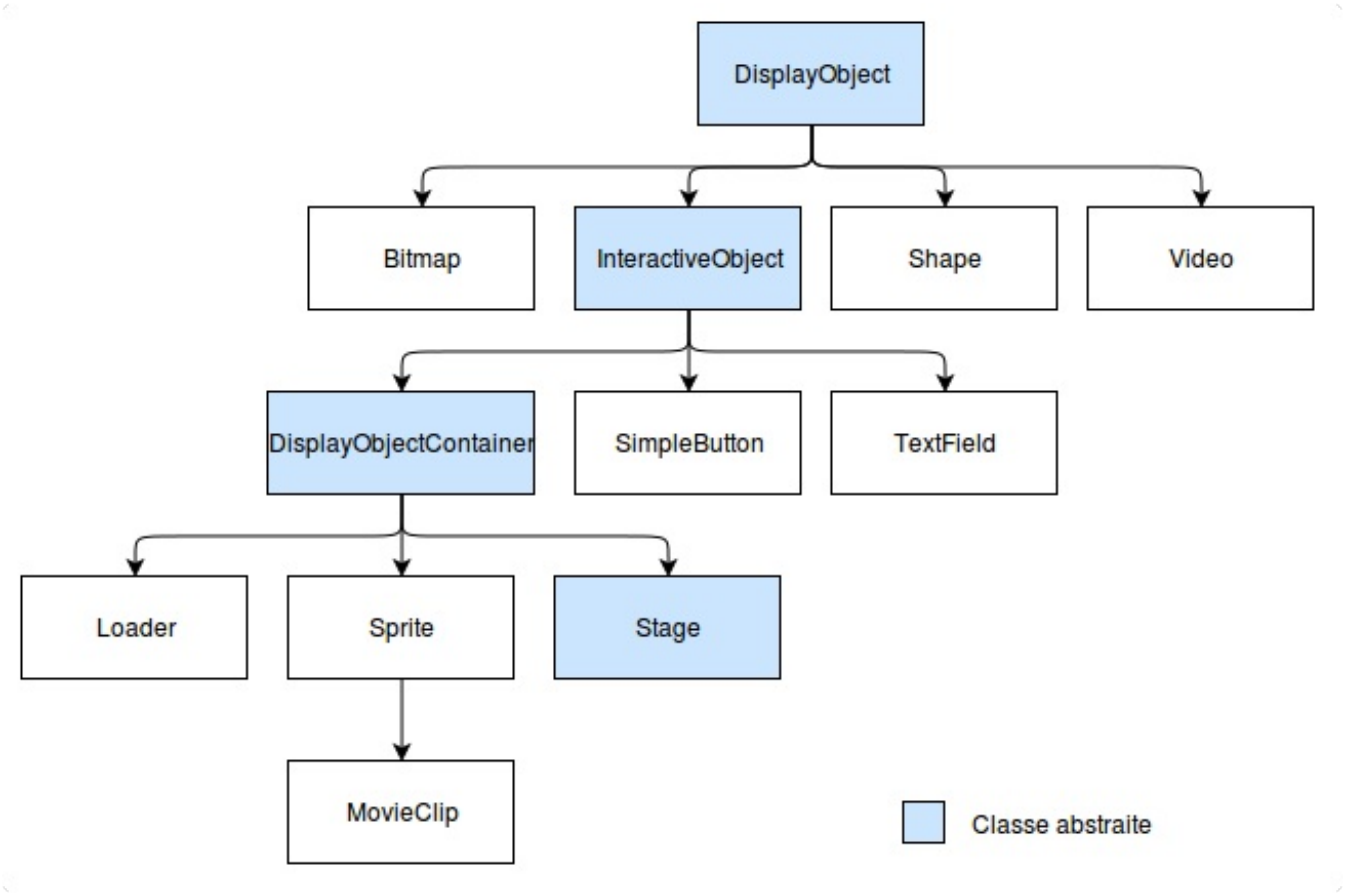


Plusieurs voitures dans l'arbre d'affichage

Il est important de se rappeler que modifier l'apparence ou la position d'un conteneur impactera tous les enfants comme s'il s'agissait d'un seul et unique objet. Il est néanmoins possible de modifier les enfants séparément.

Les classes d'affichage

En ActionScript, il existe plusieurs classes différentes dédiées à l'affichage, comme on peut le constater à la figure suivante.



Extrait de l'arbre d'héritage des classes d'affichage

Par exemple, la classe `TextField` sert à afficher du texte ; nous en reparlerons dans le prochain chapitre. Elle est une sous-classe de `InteractiveObject` (pour les objets interactifs), elle-même sous-classe de `DisplayObject`. Ainsi, toutes les classes d'affichage sont des sous-classes de la classe `DisplayObject`.

Rappel : les classes abstraites ne peuvent pas être instanciées directement, c'est-à-dire que l'on ne peut pas créer de nouveaux objets de ces classes avec le mot-clé **`new`**. Il est toutefois possible de spécifier des paramètres de type de classes abstraites dans les méthodes, ce n'est pas gênant.

La classe DisplayObject (abstraite)

Il s'agit de la classe d'affichage de base. C'est à partir d'elle que toutes les autres classes d'affichage sont dérivées ; elle contient les propriétés utiles pour manipuler des objets d'affichage simples. Encore une fois, étant donné qu'il s'agit d'une classe abstraite, inutile d'essayer de créer un objet de la classe `DisplayObject`.

La classe Bitmap

Cette classe permet d'afficher des images composées de pixels. Les objets de cette classe sont purement visuels ; il est impossible de gérer directement les clics de souris dessus par exemple. Nous apprendrons à manipuler des images dans les prochains chapitres.

La classe Shape

Cette classe permet de créer des objets légers permettant de dessiner à l'écran : tracer des lignes, des formes, remplir... À l'instar des objets de la classe `Bitmap`, les objets de la classe `Shape` sont purement visuels.

La classe Video

Comme vous vous en doutez, cette classe permet l'affichage de vidéos (avec le son). C'est aussi une classe créant des objets purement visuels.

La classe InteractiveObject (abstraite)

Cette classe permet d'introduire de **l'interactivité** pour nos objets d'affichage : il sera ainsi possible de savoir si l'utilisateur clique sur un objet d'une sous-classe de la classe `InteractiveObject`.

La classe SimpleButton

Cette classe permet de fabriquer des boutons basiques rapidement : on peut ainsi lui spécifier quel objet sera affiché dans différents états (normal, souris au-dessus, cliqué, ...), et automatiquement changer le curseur de la souris au-dessus en doigt. Cette classe est utilisée par `Flash Pro` d'Adobe pour créer les boutons.

La classe TextField

Avec cette classe, nous présenterons du texte à l'écran pendant le prochain chapitre. Il est possible de formater le texte à l'aide d'une forme très allégée d'HTML (le langage principal utilisé pour créer des sites web).

La classe DisplayObjectContainer (abstraite)

Cette dernière classe abstraite introduit la notion de **conteneur** : il est désormais possible d'ajouter plusieurs objets d'affichage enfants dans un seul conteneur, afin de les manipuler ensemble comme s'il s'agissait d'un seul objet (pensez à la fonction *Grouper* des logiciels de bureautique).

La classe Loader

Cette classe nous permettra de charger du contenu externe à notre application, **à condition qu'il soit visuel**.

La classe Stage (abstraite)

Les objets de cette classe sont des objets spéciaux qui représentent la scène d'affichage de l'application. Il est bien évidemment impossible de créer directement une instance de la classe `Stage`. Attention, la plupart des propriétés héritées des classes-mères ne fonctionnent pas sur eux (comme par exemple la position, l'opacité, etc.).

La classe Sprite

Non, il ne s'agit pas de la boisson ! Les objets de cette classe sont les plus utilisés en tant que conteneurs, et en tant que dessins interactifs. Vous en userez (et abuserez) en programmant en ActionScript !

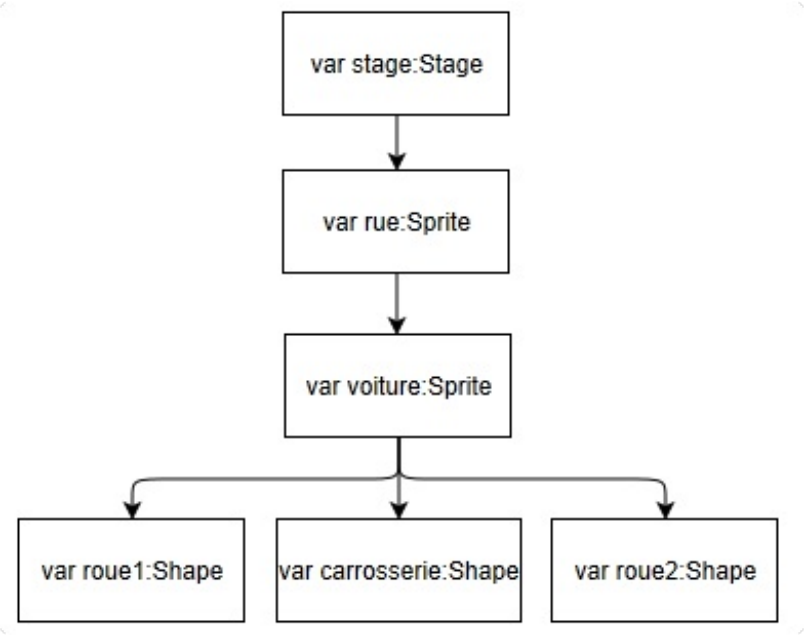
La classe MovieClip

Cette classe dérivée de la classe `Sprite` ajoute un composant complexe et plutôt lourd : une ligne de temps (ou *timeline* en anglais). Cela consiste en une série d'images-clés pouvant être réparties sur différents niveaux, afin de créer des animations complexes (voir figure suivante). Cette classe a tout d'abord été conçue pour être utilisée dans le logiciel d'animation `Flash Pro` d'Adobe ; ainsi, étant d'un intérêt limité pour une utilisation en ActionScript, nous ne l'utiliserons quasiment pas dans ce cours.

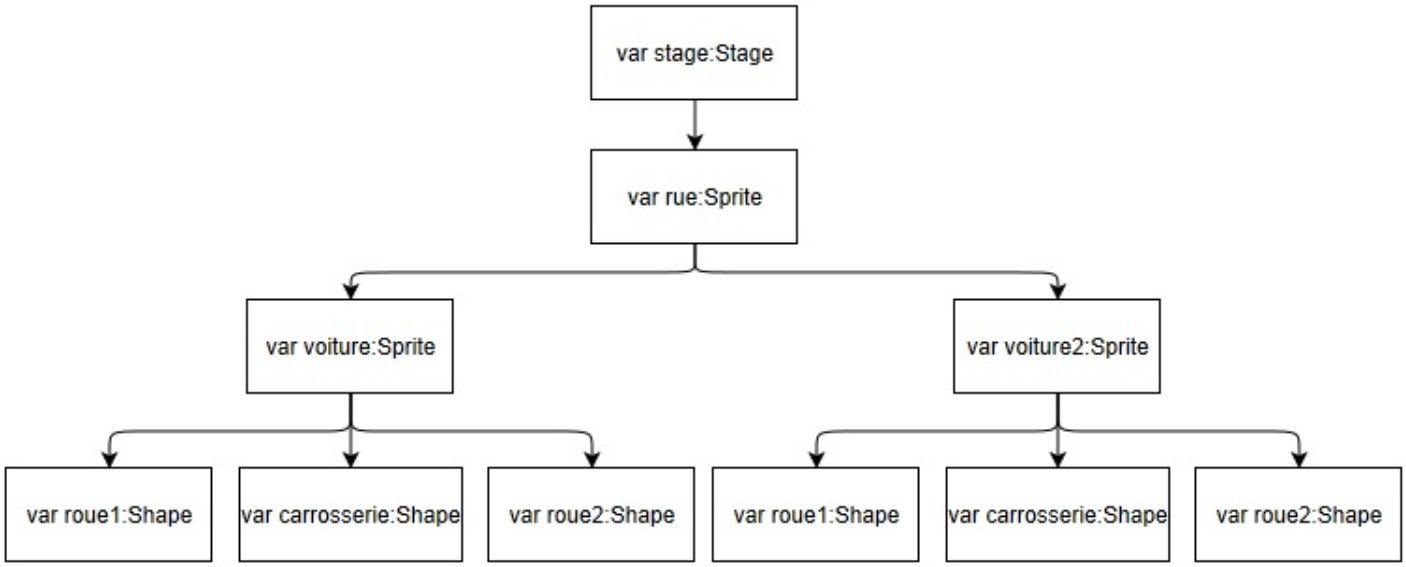
Un exemple de timeline dans Flash Pro

Il existe d'autres classes d'affichage que je n'ai pas mentionnées ici : nous ne les aborderons pas, étant trop avancées. Mais si vous êtes curieux, vous pouvez toujours explorer les possibilités supplémentaires qui s'offrent à vous en visitant la documentation.

Si nous reprenions notre exemple de la voiture, les schémas des arbres d'affichage ressembleraient aux deux figures suivantes.



L'arbre d'affichage de la voiture



Plusieurs voitures dans l'arbre d'affichage

Manipuler les conteneurs

Toutes les nouvelles propriétés (attributs et méthodes) que nous allons voir sont issues de la classe `DisplayObjectContainer`. Je vous conseille d'aller lire en parallèle la documentation officielle du langage ActionScript 3, en français et disponible à cette adresse. N'hésitez surtout pas à l'utiliser aussi souvent que vous en sentez le besoin, elle est très pratique !

Buvez du `Sprite` !

Comme nous l'avons vu précédemment, la classe `Sprite` va nous permettre de créer des conteneurs d'affichage très polyvalents tout en restant légers. Commençons par créer un conteneur que nous appellerons `voiture`.

Code : Actionscript

```
// Création du conteneur 'voiture'
var voiture:Sprite = new Sprite();
```

Comme vous pouvez le remarquer, le constructeur de la classe `Sprite` ne prend pas de paramètre.

Ajouter des enfants

Pour ajouter des objets d'affichage enfants dans notre conteneur `voiture`, il faut utiliser la méthode `addChild(enfant:DisplayObject):void` fournie par la classe `DisplayObjectContainer`. Elle prend en paramètre l'enfant à ajouter dans le conteneur.

Code : Actionscript

```
// Créons une roue
var roue:Sprite = new Sprite();
// Ajoutons-la dans la voiture
voiture.addChild(roue);

// Créons la rue et ajoutons la voiture dans la rue
var rue:Sprite = new Sprite();
rue.addChild(voiture);
```

L'objet `roue1` est désormais affiché à l'intérieur du conteneur `voiture`, lui-même affiché dans le conteneur `rue`. Mais il manque quelque chose : il faut ajouter l'objet `rue` dans la scène principale, sinon il ne sera pas affiché ! 

Afficher un objet sur la scène principale

Dans notre projet, nous utilisons une classe principale que nous avons appelée `Main` (dans le fichier `Main.as`) :

Code : Actionscript

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    /**
* ...
* @author Guillaume
*/
    public class Main extends Sprite {

        public function Main():void {
            if (stage)
                init();
            else
                addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // entry point

            trace("Hello world !");
        }

    }

}
```

Regardez droit dans les yeux la déclaration de la classe `Main` : c'est une sous-classe de `Sprite` ! Notre classe principale est donc un conteneur, qui est automatiquement ajouté à la scène au démarrage de notre application ; ainsi, il ne nous reste plus qu'à ajouter nos objets dans le conteneur de classe `Main` qui nous est offert !

Pour ajouter notre rue sur la scène principale, nous procéderons ainsi :

Code : Actionscript

```
this.addChild(rue);
```

Le code peut être raccourci en enlevant le mot-clé **this** :

Code : Actionscript

```
addChild(rue);
```

Rappel : le mot-clé **this** permet d'accéder à l'objet courant. Si nous l'utilisons dans des méthodes non statiques de la classe `Main`, nous pointerons sur l'objet qui représente notre application. Il est toutefois possible d'omettre le mot-clé **this** si cela ne pose pas de problème de conflit de noms (typiquement, lorsque vous avez à la fois un attribut dans votre classe et un paramètre du même nom dans la méthode, il faut utiliser **this** pour accéder à l'attribut de la classe). Du coup, le code ci-dessus ne fonctionnera pas dans une autre classe ; il faudrait alors utiliser une variable pointant sur l'objet de la classe `Main`.

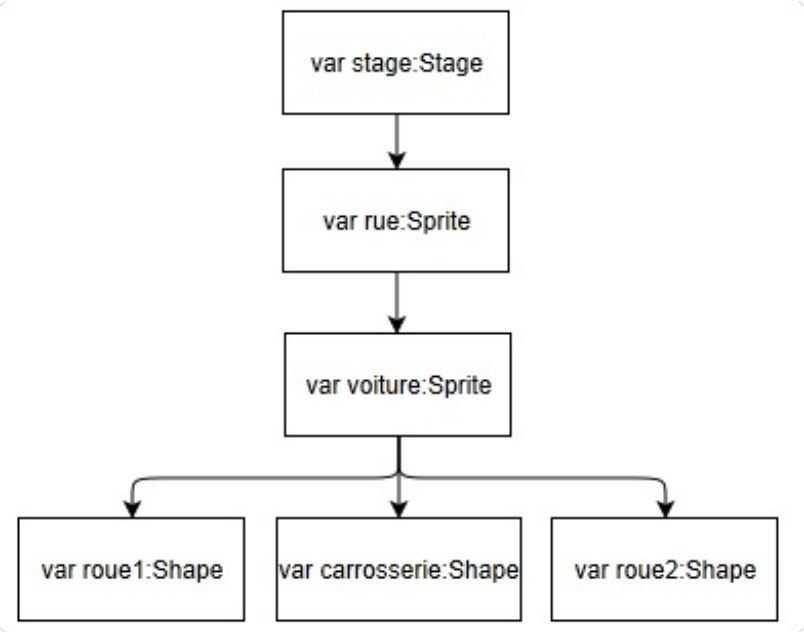
Voici le code complet de la classe `Main` :

Code : Actionscript

```
package {
    import flash.display.Sprite;
    import flash.events.Event;

    /**
* ...
* @author Guillaume
*/
    public class Main extends Sprite {

        public function Main():void {
            if (stage)
                init();
            else
                addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // entry point

            // Création du conteneur 'voiture'
            var voiture:Sprite = new Sprite();

            // Créons une roue
            var roue:Sprite = new Sprite();
            // Ajoutons-la dans la voiture
            voiture.addChild(roue);

            // Créons la rue et ajoutons la voiture dans la rue
            var rue:Sprite = new Sprite();
            rue.addChild(voiture);

            // On ajoute la rue sur la scène principale
            addChild(rue);
        }

    }

}
```

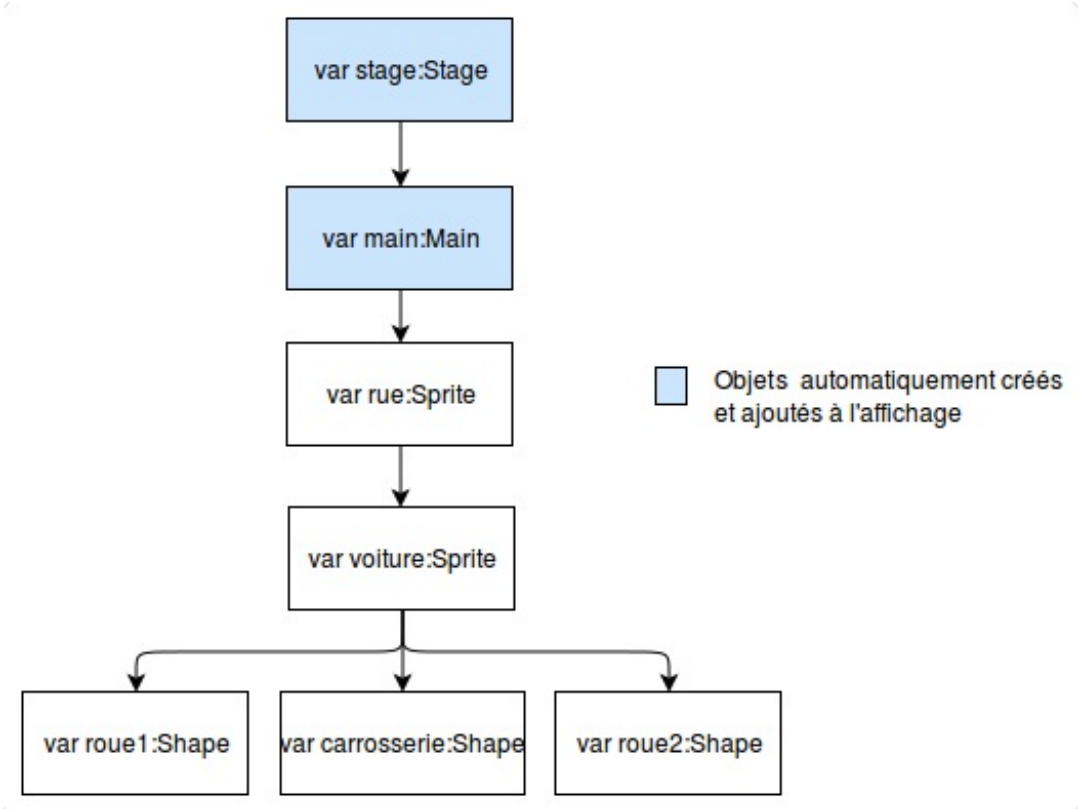
Si vous testez le code à ce stade, vous risquez d'être un peu déçus : en effet, les conteneurs sont *vides par défaut* !
Vous devrez vous contenter d'une triste scène vide, mais nous allons bientôt remédier à cela.

Reprenons maintenant le schéma représentant l'arbre d'affichage, que nous avons détaillé plus haut (voir figure suivante).



L'arbre d'affichage de la voiture

Corrigeons-le pour y inclure notre objet de la classe `Main` (voir figure suivante).



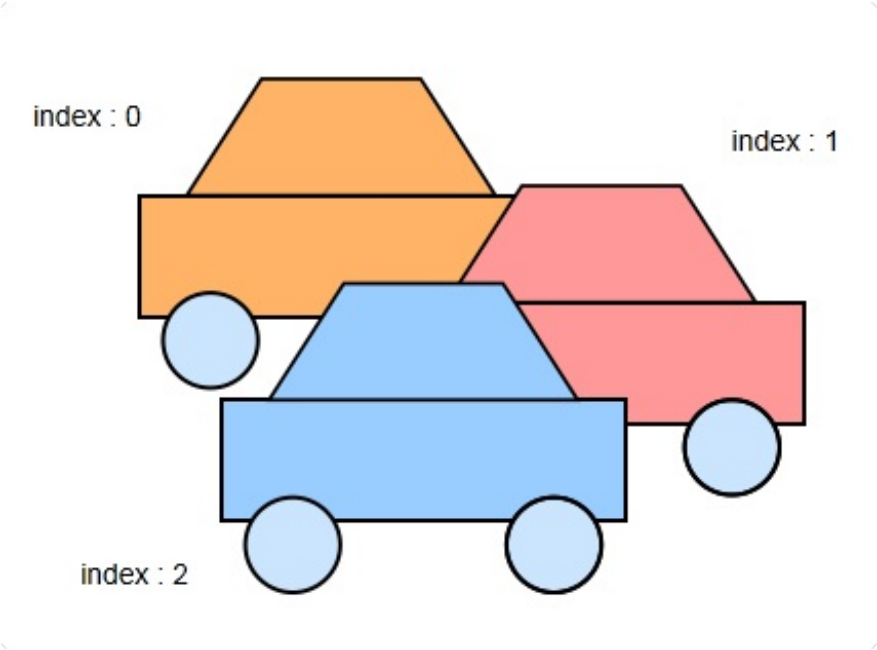
L'arbre d'affichage de notre application

Un objet ne peut être affiché à deux endroits différents ; si jamais vous l'ajoutez à un conteneur alors que l'objet est déjà dans un autre conteneur, il sera automatiquement retiré de ce dernier avant d'être ajouté dans le nouveau conteneur.

L'index d'affichage

Cette notion correspond à l'ordre d'affichage des enfants dans un conteneur. Si un objet a un index plus grand qu'un autre, il sera affiché devant, et inversement. L'index est compris entre `0` et `conteneur.numChildren - 1` (le nombre d'enfants moins un).

À la figure suivante par exemple, la voiture bleue est au-dessus des autres car elle a l'index d'affichage le plus élevé. En revanche, la voiture orange est tout en dessous car elle a un index égal à zéro.



L'index d'affichage de trois voitures superposées

Il est impossible de mettre un enfant à un index en dehors des bornes que nous avons définies : ainsi, on ne peut pas mettre un objet à un index inférieur à `0` ou supérieur à `conteneur.numChildren - 1`.

On peut obtenir l'index d'un enfant en utilisant la méthode `getChildIndex():int` (renvoyant un `int`), qui prend en paramètre l'objet enfant en question :

Code : Actionscript

```
// On crée la rue
var rue:Sprite = new Sprite();

// On ajoute trois voiture dans la rue
var voiture1:Sprite = new Sprite();
rue.addChild(voiture1);

var voiture2:Sprite = new Sprite();
rue.addChild(voiture2);

var voiture3:Sprite = new Sprite();
rue.addChild(voiture3);

trace("La voiture1 est à l'index " + rue.getChildIndex(voiture1));
// Affiche 0
trace("La voiture2 est à l'index " + rue.getChildIndex(voiture2));
// Affiche 1
trace("La voiture3 est à l'index " + rue.getChildIndex(voiture3));
// Affiche 2
```

Ajouter un enfant à un index précis

Il existe une variante de la méthode `addChild` que nous avons vue il y a peu de temps. Il s'agit de la méthode `addChildAt(enfant:DisplayObject, index:int):void`, qui prend deux paramètres : l'enfant à ajouter, puis l'index d'affichage où il faut l'ajouter.

Voici un exemple, où l'on ajoute chaque nouvelle voiture en arrière-plan :

Code : Actionscript

```
// On crée la rue
var rue:Sprite = new Sprite();

// On ajoute des voitures à la rue, en les mettant à l'arrière-plan
à chaque fois
var voiture1:Sprite = new Sprite();
rue.addChildAt(voiture1, 0);

var voiture2:Sprite = new Sprite();
rue.addChildAt(voiture2, 0);

var voiture3:Sprite = new Sprite();
rue.addChildAt(voiture3, 0);

trace("La voiture1 est à l'index " + rue.getChildIndex(voiture1));
// Affiche 2
trace("La voiture2 est à l'index " + rue.getChildIndex(voiture2));
// Affiche 1
trace("La voiture3 est à l'index " + rue.getChildIndex(voiture3));
// Affiche 0
```

Comme vous pouvez le constater, l'index d'affichage de chaque objet varie, car il représente à tout instant la position en profondeur de celui-ci ; si jamais un objet passe derrière un autre par exemple, son index sera automatiquement modifié en conséquence !

Opérations sur les enfants

Nombre d'enfants

Pour obtenir le nombre d'enfants que contient un conteneur, vous pouvez utiliser le **getter** `numChildren` de type `int`.

Attention, c'est un attribut en lecture seule car aucun setter n'est défini dans la classe `DisplayObjectContainer` !

Code : Actionscript

```
// On crée une rue
var rue:Sprite = new Sprite();

// On ajoute trois voitures dans la rue
var voiture1:Sprite = new Sprite();
rue.addChild(voiture1);
var voiture2:Sprite = new Sprite();
rue.addChild(voiture2);
var voiture3:Sprite = new Sprite();
rue.addChild(voiture3);

trace("Il y a " + rue.numChildren + " voitures dans la rue."); //
Affiche 3.
```

Accéder au parent d'un objet d'affichage

L'attribut `parent` permet d'accéder au conteneur parent qui contient l'objet d'affichage. Cette propriété est notamment utile à l'intérieur d'une classe, quand on ne peut pas savoir quel est le conteneur.

Code : Actionscript

```
trace(rue == voiture1.parent); // Affiche true

trace("Nombre de voitures : " + rue.numChildren); // Affiche 3
trace("Nombre de voitures : " + voiture1.parent.numChildren); //
Affiche 3 également
```

Modifier l'index d'un enfant

Une fois un objet d'affichage enfant ajouté à un conteneur, il est possible de modifier son index d'affichage. Pour cela, nous utiliserons la méthode `setChildIndex(enfant:DisplayObject, index:int):void` qui prend en paramètre l'enfant en question puis son nouvel index.

Code : Actionscript

```
// On met successivement les voitures au premier-plan
rue.setChildIndex(voiture1, 2);
rue.setChildIndex(voiture2, 2);
rue.setChildIndex(voiture3, 2);

trace("La voiture1 est à l'index " + rue.getChildIndex(voiture1));
// Affiche 0
trace("La voiture2 est à l'index " + rue.getChildIndex(voiture2));
// Affiche 1
trace("La voiture3 est à l'index " + rue.getChildIndex(voiture3));
// Affiche 2
```

Rappel : la valeur de l'index doit être située entre `0` et `conteneur.numChildren - 1` !

Échanger les index d'affichage de deux enfants

Il existe deux méthodes de la classe `DisplayObjectContainer` pour échanger la profondeur de deux objets enfants : `swapChildren()` qui prend en paramètre deux références des enfants du conteneur et `swapChildrenAt()` qui prend en paramètre deux index différents à échanger.

Code : Actionscript

```
rue.swapChildren(voiture1, voiture2);
trace("La voiture1 est à l'index " + rue.getChildIndex(voiture1));
// Affiche 1
trace("La voiture2 est à l'index " + rue.getChildIndex(voiture2));
// Affiche 0
// La voiture 1 est affichée devant la voiture 2

rue.swapChildrenAt(0, 1);
trace("La voiture1 est à l'index " + rue.getChildIndex(voiture1));
// Affiche 0
trace("La voiture2 est à l'index " + rue.getChildIndex(voiture2));
// Affiche 1
// La voiture 1 est affichée en dessous de la voiture 2
```

Déterminer si un objet est enfant d'un conteneur

Une méthode très pratique de la classe `DisplayObjectContainer` renvoie un booléen pour le savoir : j'ai nommé `contains()` ! Elle prend en paramètre un objet de la classe `DisplayObject` et retourne **`true`** si cet objet est dans la liste d'affichage du conteneur (y compris parmi les petits-enfants, parmi les enfants des petits-enfants, etc.), ou s'il s'agit du conteneur lui-même (nous considérons que le conteneur se contient lui-même). Sinon, elle renvoie **`false`**.

Code : Actionscript

```
var sprite1:Sprite = new Sprite();
var sprite2:Sprite = new Sprite();
var sprite3:Sprite = new Sprite();
var sprite4:Sprite = new Sprite();

sprite1.addChild(sprite2);
sprite2.addChild(sprite3);

trace(sprite1.contains(sprite1)); // Affiche: true
trace(sprite1.contains(sprite2)); // Affiche: true
trace(sprite1.contains(sprite3)); // Affiche: true
trace(sprite1.contains(sprite4)); // Affiche: false
```

Retirer des enfants

Pour retirer un enfant d'un parent, la méthode `removeChild(enfant:DisplayObject):void` est toute indiquée ! À l'instar de la méthode `addChild()`, cette fonction prend en paramètre l'enfant à enlever de l'affichage.

Code : Actionscript

```
// On enlève les trois voitures de la rue
rue.removeChild(voiture1);
rue.removeChild(voiture2);
rue.removeChild(voiture3);
```

Il est également possible de supprimer un enfant à un certain index, sans savoir précisément duquel il s'agit, à l'aide de la méthode
`removeChildAt(index:int):void`. Le paramètre que nous passons correspond à l'index de l'enfant que nous souhaitons enlever.

Code : Actionscript

```
// On crée la rue
var rue:Sprite = new Sprite();

// On ajoute des voitures à la rue, en les mettant à l'arrière-plan
à chaque fois
var voiture1:Sprite = new Sprite();
rue.addChildAt(voiture1, 0);

var voiture2:Sprite = new Sprite();
rue.addChildAt(voiture2, 0);

var voiture3:Sprite = new Sprite();
rue.addChildAt(voiture3, 0);

trace("La voiture1 est à l'index " + rue.getChildIndex(voiture1));
// Affiche 2
trace("La voiture2 est à l'index " + rue.getChildIndex(voiture2));
// Affiche 1
```

```
trace("La voiture3 est à l'index " + rue.getChildIndex(voiture3));
// Affiche 0

// On enlève a voiture le plus au fond
rue.removeChildAt(0); // voiture3 enlevée !

// On peut enlever les deux autres de la même manière
rue.removeChildAt(0); // voiture2 enlevée !
rue.removeChildAt(0); // voiture1 enlevée !
```

Et si on souhaite enlever tous les enfants du conteneur, n'y a-t-il pas un moyen moins fastidieux ?

La réponse est oui, grâce à la méthode `removeChildren(beginIndex:int = 0, endIndex:int = 0x7fffffff):void` qui permet de supprimer plusieurs enfants d'un coup d'un seul ! Les deux paramètres facultatifs spécifient à partir de quel index on supprime, et jusqu'à quel index. Par défaut, tous les enfants sont supprimés, donc si c'est ce que vous souhaitez faire, laissez les parenthèses vides. 

Code : Actionscript

```
rue.removeChildren(); // Toutes les voitures sont enlevées d'un coup
!
```

Propriétés utiles des objets d'affichage

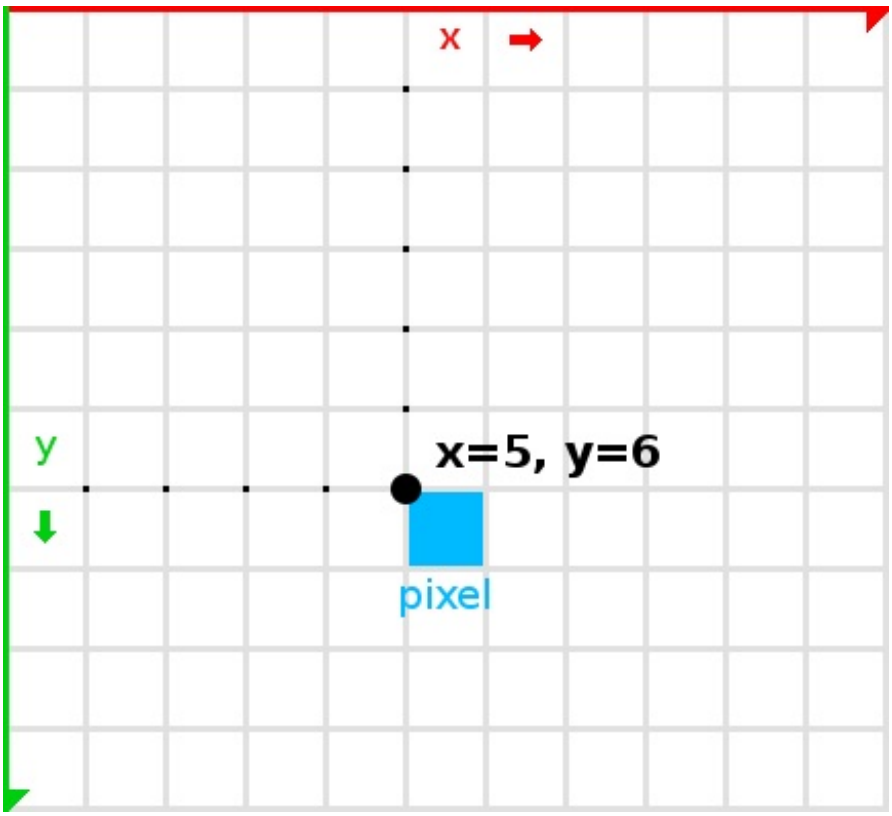
Nous allons maintenant aborder des propriétés communes à tous les objets d'affichage, qui s'avèreront très utiles ! Toutefois, nous nous limiterons pour l'instant à l'affichage 2D dans le repère mentionné en début de chapitre.

Les attributs et méthodes que nous allons voir sont des propriétés de la classe `DisplayObject`, dont la documentation est disponible ici.

Position

Tout d'abord, commençons par la position de l'objet d'affichage : deux attributs, `x` et `y`, permettent de contrôler respectivement la position horizontale et verticale de l'origine d'un objet d'affichage.

Reprenons l'exemple de l'introduction, visible à la figure suivante.



Exemple de position sur l'écran

Pour déplacer l'origine d'un objet à cet endroit précis (5, 6), nous écrirons ceci :

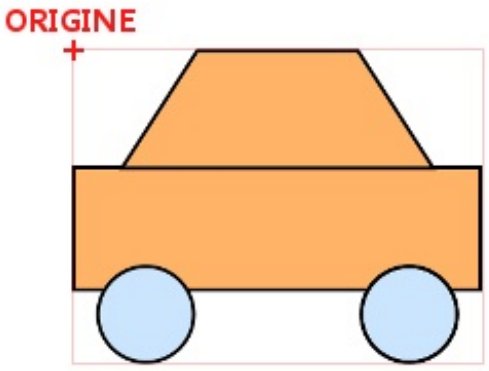
Code : Actionscript

```
var monObjet:Sprite = new Sprite();
// Modifions la position de l'objet !
monObjet.x = 5;
monObjet.y = 6;
addChild(monObjet);
```

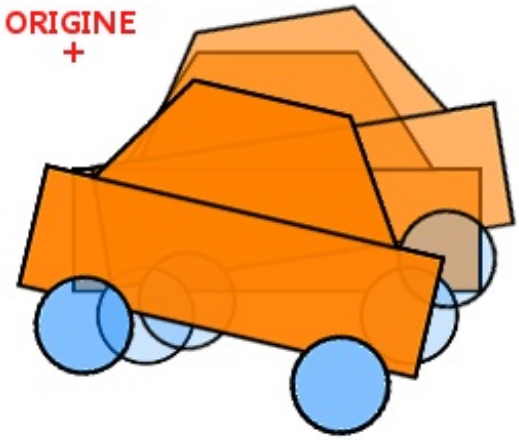
La position par défaut de tout nouvel objet d'affichage est (0, 0), c'est-à-dire que `x` vaut `0` et que `y` vaut `0` si vous n'y touchez pas.

Un mot sur l'origine

L'origine d'un objet d'affichage est le point qui représente sa position actuelle. Par défaut, elle est située en haut à gauche (voir figure suivante), et elle ne peut pas être déplacée.



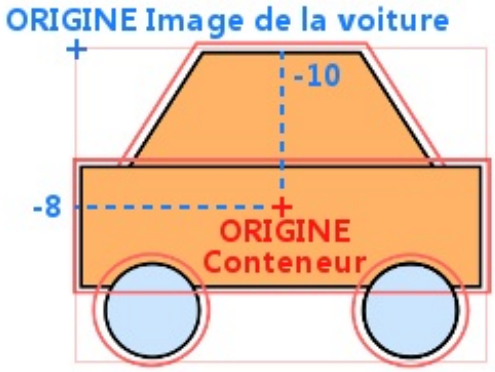
L'origine est en haut à gauche par défaut



Rotation du conteneur avec l'origine par défaut

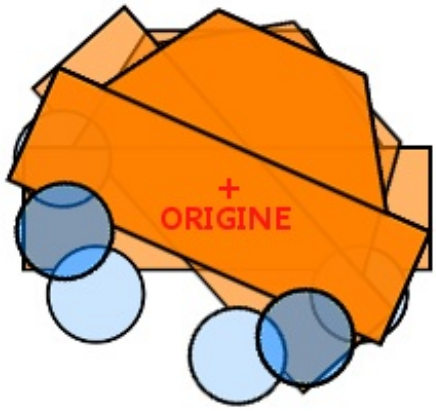
Comment modifier l'origine de notre conteneur si on ne peut pas la déplacer ?

Et bien, c'est très simple, il suffit de déplacer les enfants à l'intérieur de notre conteneur, pour donner l'illusion de modifier l'origine (voir figure suivante) !



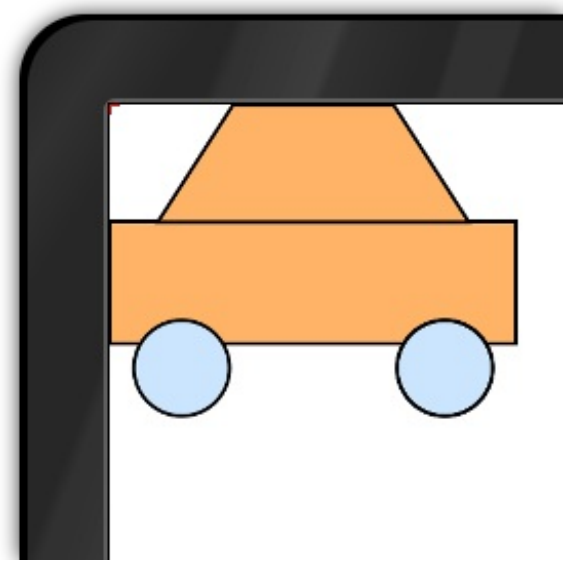
On déplace l'image de voiture à l'intérieur du conteneur

Et c'est gagné (voir figure suivante) !

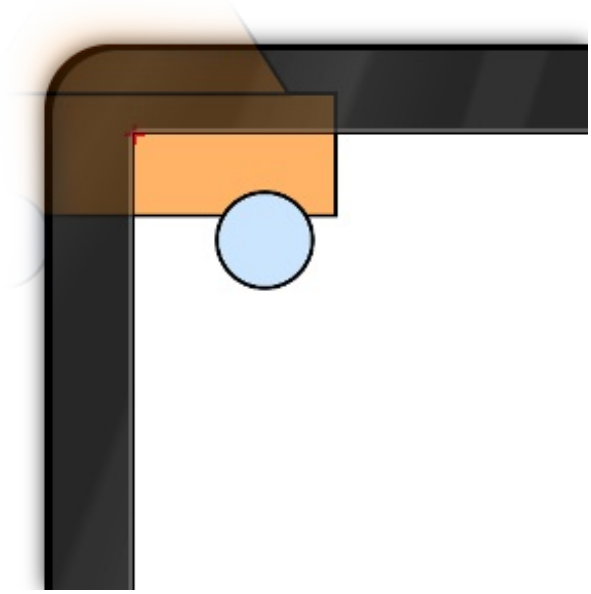


Rotation du conteneur avec la nouvelle origine

L'affichage à l'écran de la voiture à la position (0, 0) ressemblerait alors à la figure suivante.



Dans le cas de l'origine par défaut



Dans le cas où l'on a centré l'origine

Par exemple, si vous ajoutez un objet d'affichage `voiture` dans votre conteneur `rue`, et que vous positionnez la voiture en négatif à 100 pixels de l'origine de la rue, une partie de votre voiture sera très certainement en dehors de l'écran. Mais l'origine de la rue n'a pas changé entre-temps : elle est toujours à sa position de départ (0, 0). Maintenant, vous pouvez faire réapparaître la voiture en entier en déplaçant la rue de 100 pixels vers la droite ! Son origine va ainsi changer et devenir (100, 0), ainsi la position de la voiture par rapport à la scène sera (0, 0) ! Tout est relatif ! 

Taille

Taille absolue

La taille absolue d'un objet d'affichage, c'est-à-dire la taille qu'il prend à l'écran, peut être lue ou modifiée à l'aide des attributs `width` (longueur) et `height` (hauteur), en pixels. La taille est toujours exprimée en valeurs positives.

Code : Actionscript

```
var maVoiture:Voiture= new Voiture();
// Modifions la taille de la voiture pour qu'elle fasse 100x100
pixels !
maVoiture.width = 100;
maVoiture.height = 100;
addChild(maVoiture);
```

Modifier la taille d'un conteneur vide et sans dessin n'a aucun sens, et ne donnera aucun résultat. Ses deux attributs `width` et `height` seront toujours égaux à `0`.

Taille relative

On peut également redimensionner un objet d'affichage à l'aide de pourcentages, grâces aux attributs `scaleX` pour redimensionner en longueur, et `scaleY` en hauteur. Les pourcentages ne sont pas directement exprimés en tant que tel : par exemple, `scaleX = 1` signifie que la longueur de l'objet est à 100%, `scaleY = 2` signifie que l'objet est deux fois plus haut que sa taille absolue d'origine et `scaleY = 0.5` signifie que l'objet est moitié moins haut.

Code : Actionscript

```
var maVoiture:Voiture= new Voiture();
// Modifions la taille de la voiture pour qu'elle fasse 100x100
pixels !
maVoiture.width = 100;
maVoiture.height = 100;

// Réduisons la taille de la voiture de moitié !
maVoiture.scaleX = 0.5;
maVoiture.scaleY = 0.5;
trace(maVoiture.width); // Affiche: 50
trace(maVoiture.height); // Affiche: 50

// Remettons-là à sa taille normale :
maVoiture.scaleX = 1;
maVoiture.scaleY = 1;
trace(maVoiture.width); // Affiche: 100
trace(maVoiture.height); // Affiche: 100
addChild(maVoiture);
```

Avec ses propriétés, il est possible d'inverser un objet d'affichage à l'aide de valeurs négatives :

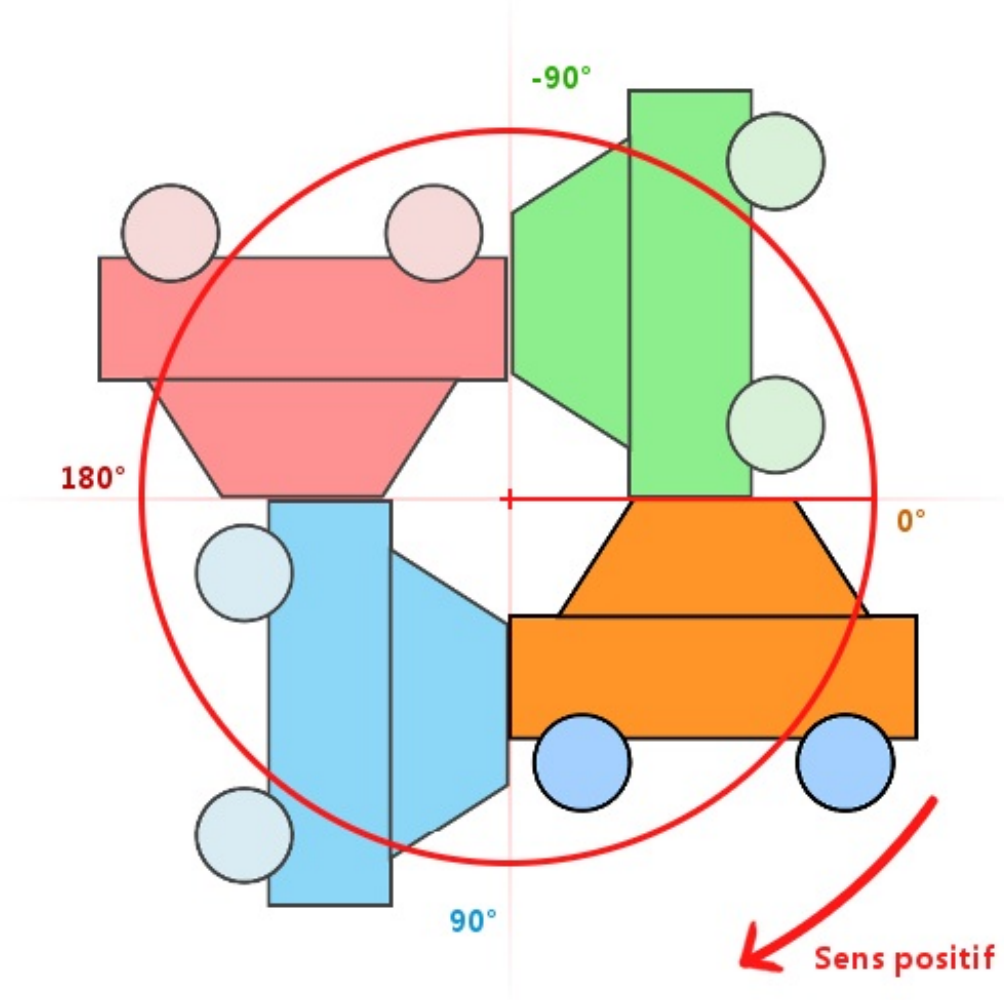
Code : Actionscript

```
var maVoiture:Voiture= new Voiture();
// Modifions la taille de la voiture pour qu'elle fasse 100x100
pixels !
maVoiture.width = 100;
maVoiture.height = 100;

// Inversons horizontalement la voiture !
maVoiture.scaleX = -1;
trace(maVoiture.width); // Affiche: 100
trace(maVoiture.height); // Affiche: 100
addChild(maVoiture);
```

Rappel : les attributs de taille `width` et `height` sont toujours positifs.

Rotation

L'attribut `rotation` permet de faire tourner des objets d'affichage autour de leur origine (voir figure suivante), *en utilisant des degrés*. Les valeurs comprises entre 0 et 180 représentent la rotation en sens horaire ; les valeurs comprises entre 0 et -180 représentent la rotation en sens antihoraire. Les valeurs hors de cette plage sont alors réajustées (ajoutées ou soustraites de 360) pour obtenir une valeur comprise dans la plage. Par exemple, l'instruction `voiture.rotation = 400` correspond à `voiture.rotation = 40` (car 400 - 360 = 40).



Rotation d'un objet (avec l'origine par défaut)

Si vous voulez utiliser des valeurs en radians, il ne faut pas oublier de les convertir en degrés avant d'utiliser l'attribut `rotation` : on écrira par exemple `voiture.rotation = maValeurEnRadians / Math.PI * 180;` en utilisant la constant `Math.PI` qui contient une valeur assez précise du nombre pi.

Transparence

Visibilité

L'attribut `visible` vaut **`true`** si l'objet est visible, **`false`** s'il est invisible. Par défaut, sa valeur est **`true`** (l'objet est visible).

Code : Actionscript

```
var maVoiture:Voiture= new Voiture();
// Rien ne sera affiché à l'écran
maVoiture.visible = false;
addChild(maVoiture);
```

Un objet d'affichage avec une visibilité à faux n'est pas du tout rendu à l'écran, et ne peut donc pas interagir avec la souris par exemple. Cela permet toutefois d'économiser des ressources de l'ordinateur pour améliorer les performances. Un bon conseil : si un objet doit être invisible, utilisez cette propriété à chaque fois que vous le pouvez !

Opacité

L'attribut `alpha` de type `Number` détermine l'opacité d'un objet d'affichage, c'est-à-dire s'il est plus ou moins transparent. Les valeurs vont de 0 (invisible) à 1 (opaque).

Code : Actionscript

```
var maVoiture:Sprite = new Sprite();
// La voiture sera à moitié transparente
maVoiture.alpha = 0.5;
addChild(maVoiture);
```

L'opacité d'un conteneur est un multiplicateur de l'opacité de tous ces enfants, et ainsi de suite. Par exemple, si un enfant a une opacité de 0.5, et que le conteneur a une opacité de 0.5, l'opacité réelle de l'enfant sera de 0.5 x 0.5 = 0.25 : l'enfant est donc à trois quarts transparent.

Code : Actionscript

```
var enfant:Sprite = new Voiture();
// L'enfant sera à moitié transparent
enfant.alpha = 0.5;

var conteneur:Sprite = new Sprite();
// Le conteneur sera aussi à moitié transparent
conteneur.alpha = 0.5;
conteneur.addChild(maVoiture);

// Au final, l'opacité de l'enfant vaut 0.25
```

Il est possible de mettre une valeur supérieure à 1 : cela aura pour effet d'augmenter l'opacité des enfants de l'objet. Par exemple, si un enfant a pour opacité 0.5, et que l'on met l'opacité de son conteneur à 2, l'opacité de l'enfant réelle sera 0.5 x 2 = 1 : l'enfant est donc entièrement opaque.

Un objet visible mais transparent (c'est-à-dire que sa visibilité est à vrai mais que son opacité est à 0) est quand même rendu à l'écran, et donc consomme des ressources de l'ordinateur. Toutefois, il reste interactif, il est possible de cliquer dessus par exemple.

Supprimer un objet d'affichage de la mémoire

À ce stade du cours, pour supprimer un objet d'affichage de la mémoire, il est nécessaire de respecter les points suivants :

- l'objet ne doit pas faire partie de la liste d'affichage, c'est-à-dire qu'il ne doit pas être l'enfant d'un conteneur ;
- il ne doit plus y avoir une seule référence à cet objet.

Nous étofferons cette liste au fur-et-à-mesure du cours, lorsque nous aborderons de nouvelles notions, afin que vous soyez experts en mémoire non-saturée ! 

Pour supprimer un objet de la liste d'affichage, vous pouvez utiliser la méthode `removeChild()` de la classe `DisplayObjectContainer` :

Code : Actionscript

```
var maVoiture:Voiture = new Voiture();
rue.addChild(maVoiture);
// Supprimons la voiture de la liste d'affichage
rue.removeChild(maVoiture);
```

L'objet en question doit être un enfant du conteneur, sinon, une erreur sera envoyée à l'appel de la méthode `removeChild()`.

Si jamais vous voulez supprimer l'objet d'affichage depuis sa propre classe, il faut utiliser l'attribut `parent` et ruser un peu :

Code : Actionscript

```
package
{
    import flash.display.Sprite;

    public class Voiture extends Sprite
    {
        public function Voiture()
        {

        }

        public function removeFromParent():void
        {
            // Vérifions que la voiture est sur la liste
d'affichage (cad. que le parent existe) avant d'appeler
removeChild()
            if (parent != null)
            {
                // Supprimons l'objet de la liste d'affichage
                parent.removeChild(this);
            }
        }
    }

}
```

Mais comment supprimer une référence à un objet ?

C'est très simple : il suffit d'affecter la variable en question avec le mot-clé **`null`** :

Code : Actionscript

```
var maVoiture:Voiture = new Voiture();
rue.addChild(maVoiture);
// Supprimons la voiture de la liste d'affichage
rue.removeChild(maVoiture);
// Supprimons la référence à la voiture !
maVoiture = null;
```

À ce stade, l'objet de la classe `Voiture` n'a pas encore été supprimé : il est encore dans la mémoire de votre ordinateur. Il sera effectivement supprimé lorsque le *ramasse-miette* (*garbage-collector* en anglais) sera passé pour nettoyer la mémoire. Seulement, pour que le ramasse-miette puisse faire son travail, et déterminer quels objets doivent être supprimés, il faut respecter les points énumérés ci-dessus. On dit alors que ces objets sont *éligibles au nettoyage* par le ramasse-miette.

Vous remarquerez ainsi qu'il n'existe aucun moyen de supprimer un objet directement de la mémoire, sauf pour quelques rares exceptions. La plupart du temps, il faut laisser le ramasse-miette le faire à notre place. Cela marche donc pour tous les objets, pas seulement les objets d'affichage ; mais dans ce cas, inutile d'essayer de retirer de la liste d'affichage des objets qui n'ont rien à voir ! Supprimer leurs références suffit pour l'instant.

En résumé

- En ActionScript 3, les couleurs sont décrites en notation hexadécimale, précédées des caractères `0x` ('zéro' et 'x').
- L'affichage à un écran se fait dans un repère partant du coin supérieur gauche de l'écran. Les coordonnés en x vont de gauche à droite, et en y de haut en bas.
- Les objets d'affichage sont contenus dans un arbre : ils peuvent contenir chacun des enfants pouvant eux-même contenir

d'autres enfants.
- On peut manipuler les enfants des conteneurs à l'aide des propriétés de la classe `DisplayObjectContainer`.
- Une multitude de propriétés de la classe `DisplayObject` permettent de manipuler les objets d'affichage.
- Il faut respecter quelques consignes pour qu'un objet d'affichage soit supprimé de la mémoire par le ramasse-miette.

Afficher du texte

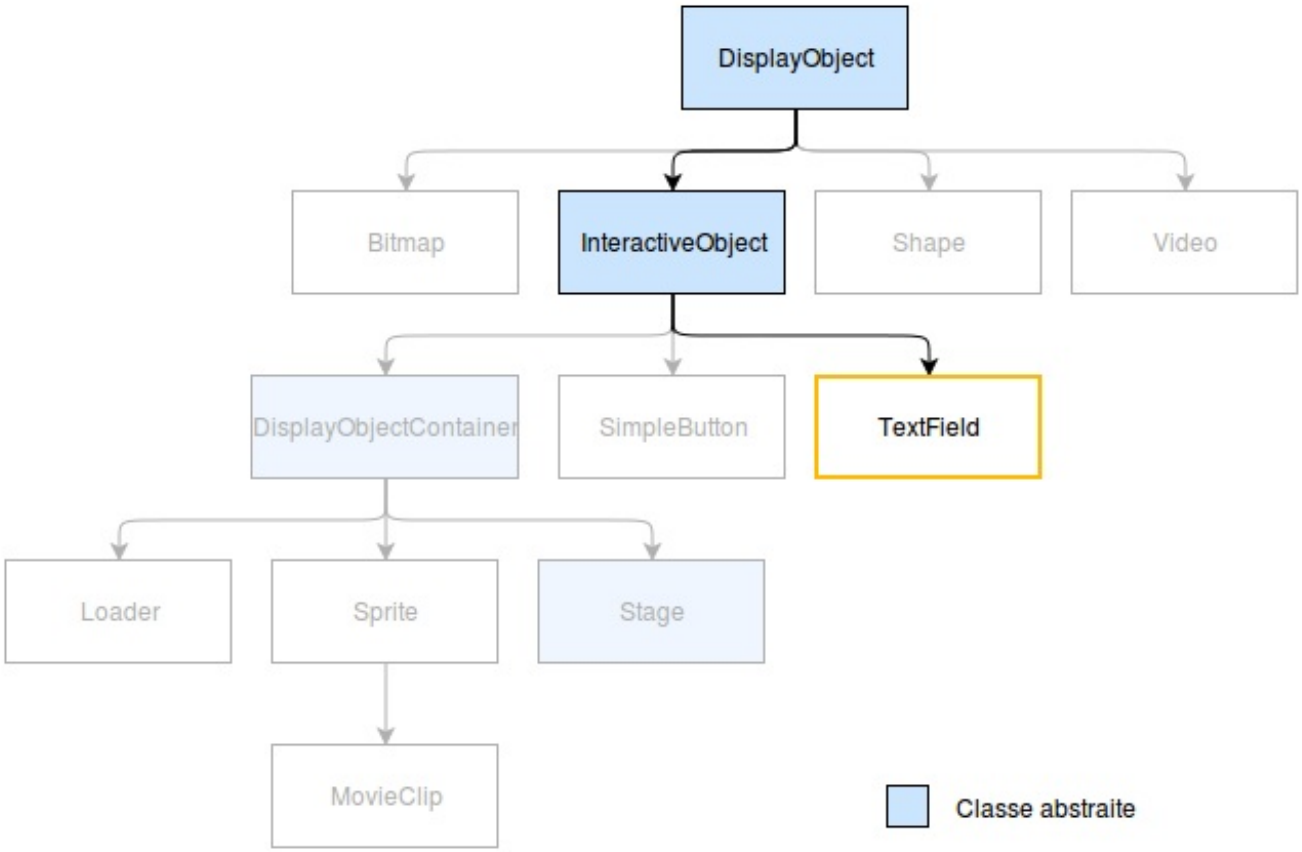
Mesdames, messieurs, j'ai l'honneur de vous présenter le premier chapitre où nous allons pouvoir afficher quelque chose sur notre fenêtre désespérément vide !
Et il s'agit rien de moins que du texte, grâce à une classe que nous allons retourner dans tous les sens : `TextField`.

Que le spectacle commence !

Une histoire de TextField

La classe

Comme vous l'aurez sûrement compris, nous allons utiliser dans ce chapitre la classe `TextField`. Pour se situer, reprenons le diagramme des classes des objets d'affichage :



TextField parmi l'arbre des classes d'affichage

Comme vous pouvez le voir, la classe `TextField` est une sous-classe de `InteractiveObject`, elle hérite donc de toutes les fonctionnalités qui permettent à l'utilisateur d'interagir avec les objets `TextField` (clic, sélection de texte, frappe au clavier, etc.). Mais c'est aussi un objet d'affichage comme les autres, car la classe `InteractiveObject` est fille de la classe `DisplayObject` : nous pouvons donc bouger, redimensionner ou tourner nos champs de texte comme bon nous semble, voir les rendre semi-transparents !

Comme d'habitude, je vous conseille de lire la documentation sur la classe `TextField`, disponible ici.

Utilisation de base

Créer un champ de texte

Comme toutes les classes des objets d'affichage, la classe `TextField` dispose d'un constructeur sans paramètre. Nous allons donc l'utiliser pour créer des champs de texte comme suit :

Code : Actionscript

```
var monSuperTexte:TextField = new TextField();
```

Du texte brut

L'attribut `text` contient le texte brut (c'est-à-dire non formaté) qui est affiché dans le champ de texte. Par défaut, le texte est vide, c'est-à-dire que l'attribut contient une chaîne de caractères vide : `""`.

Il s'utilise comme n'importe quel autre attribut :

Code : Actionscript

```
// Affectation d'un texte brut
monSuperTexte.text = 'Hello World!';

// Avec une variable :
var maSuperChaine:String = 'Je m\'apelle Zozor !';
monSuperTexte.text = maSuperChaine;

// Lecture du contenu brut du champ de texte :
trace('Zozor a dit : "' + monSuperTexte.text + '"'); // Affiche:
Zozor a dit : "Je m'appelle Zozor !"
```

À ne pas oublier !

Pour que notre champ de texte soit affiché, il ne faut surtout pas oublier de l'ajouter sur la liste d'affichage, c'est-à-dire qu'il faut l'ajouter parmi les enfants d'un conteneur ! Ainsi, dans notre classe `Main`, écrivons ceci :

Code : Actionscript

```
// Ajoutons le champ de texte sur la scène principale :
addChild(monSuperTexte);
```

Rappel : la classe `Main` est une sous-classe de la classe `Sprite`, elle-même une sous-classe de la classe `DisplayObjectContainer` ! Nous pouvons donc y ajouter des objets d'affichage (nous serions bien embêtés sinon).

Le résultat

Par défaut, la couleur du texte est le noir et sans décoration (bordure ou fond), la police utilisée est une police par défaut de votre système (*Times New Roman* sur Windows/Linux et *Times* sur Mac), la taille est de 12 points et le texte n'est pas éditable. Mais ne vous inquiétez pas, tout ceci est entièrement paramétrable, comme certains ont pu le remarquer en lisant la documentation. 

Pour que vous puissiez vérifier votre code, voici l'ensemble de la classe `Main` permettant d'afficher du texte à l'écran :

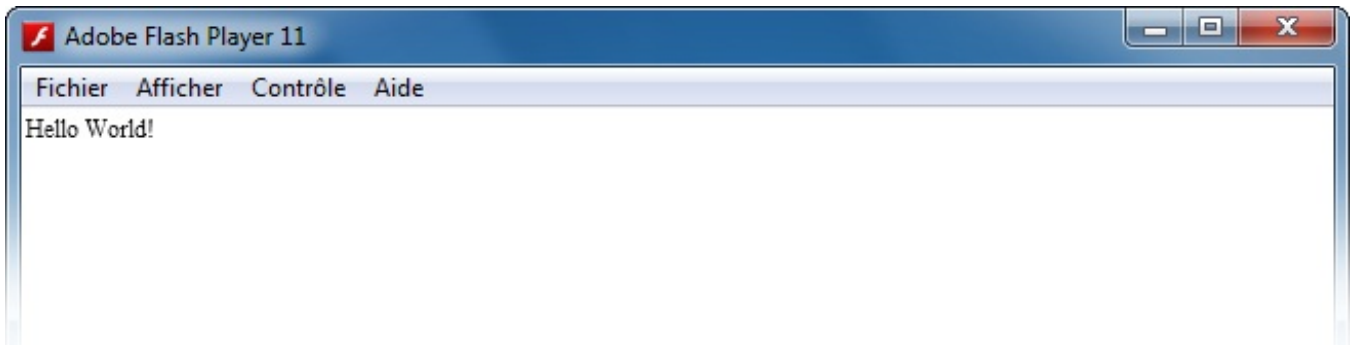
Code : Actionscript

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.text.TextField;

    /**
* Cette application sait afficher du texte !
*/
```

```
        [SWF(width="800",height="600",backgroundColor="#ffffff")]

        public class Main extends Sprite
        {

            public function Main():void
            {
                if (stage)
                    init();
                else
                    addEventListener(Event.ADDED_TO_STAGE, init);
            }

            private function init(e:Event = null):void
            {
                removeEventListener(Event.ADDED_TO_STAGE, init);
                // entry point

                // Mon premier champ de texte ! \o/
                var monSuperTexte:TextField = new TextField();
                monSuperTexte.text = 'Hello World!';
                addChild(monSuperTexte);
            }
        }

    }
```

Pour que vous puissiez voir quelque chose, vérifiez que la couleur de fond de votre animation est en blanc : `[SWF(width="800",height="600",backgroundColor="#ffffff")]`. Du texte noir sur fond noir par exemple, ce n'est pas ce qu'il y a de plus lisible.

La figure suivante vous montre ce que cela donne à l'écran.



Notre premier champ de texte !

Bon, j'avoue que ce n'est pas très joli, mais nous allons améliorer cela dans peu de temps.

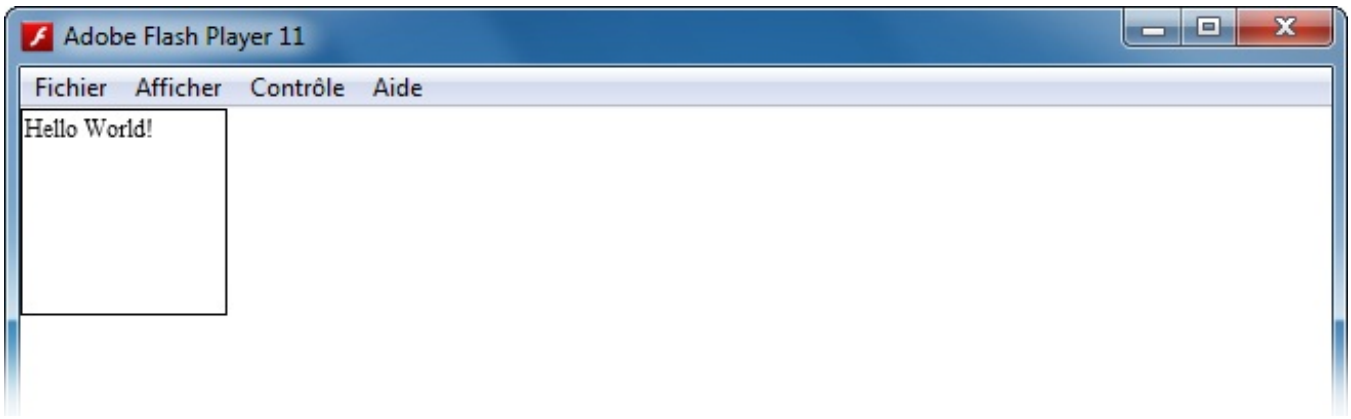
La taille par défaut d'un champ de texte est 100 pixels de largeur et 100 pixels de hauteur. Il est bien entendu possible de la modifier.

Pour le vérifier visuellement, nous pouvons afficher une bordure autour du champ de texte :

Code : Actionscript

```
    monSuperTexte.border = true;
```

Ce qui donne à l'écran la figure suivante.



Le champ de texte, avec sa bordure

Ce n'est pas ce qu'il y a de plus esthétique, alors enlevons la bordure en commentant la ligne que nous venons d'ajouter :

Code : Actionscript

```
// monSuperTexte.border = true;
```

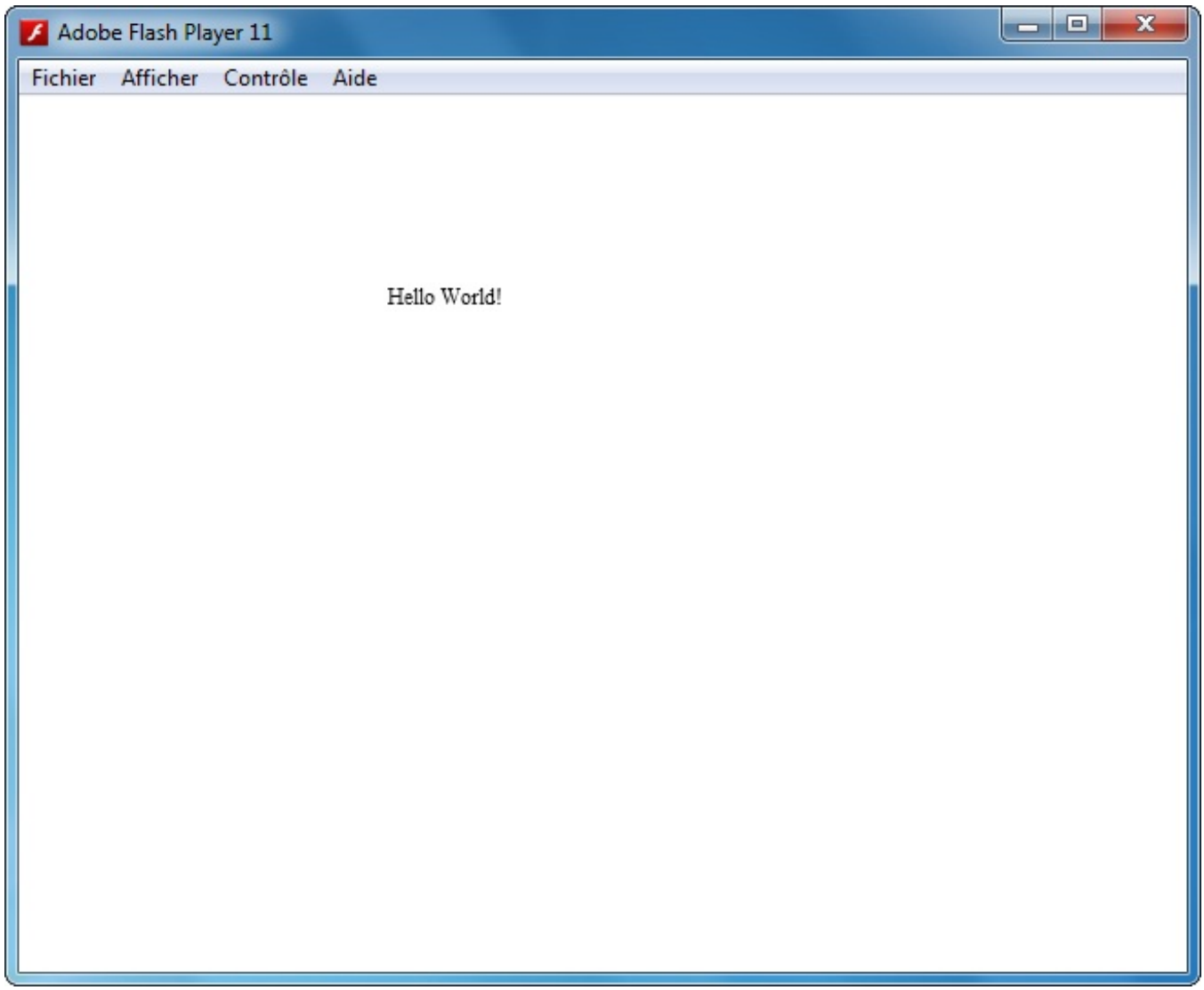
Exerçons-nous maintenant à utiliser les propriétés des objets d'affichage, maintenant que nous en avons un vrai sous la main !

Commençons par déplacer notre champ de texte à la position (200, 100) :

Code : Actionscript

```
monSuperTexte.x = 200;
monSuperTexte.y = 100;
```

Vous devriez maintenant avoir quelque chose ressemblant à la figure suivante.



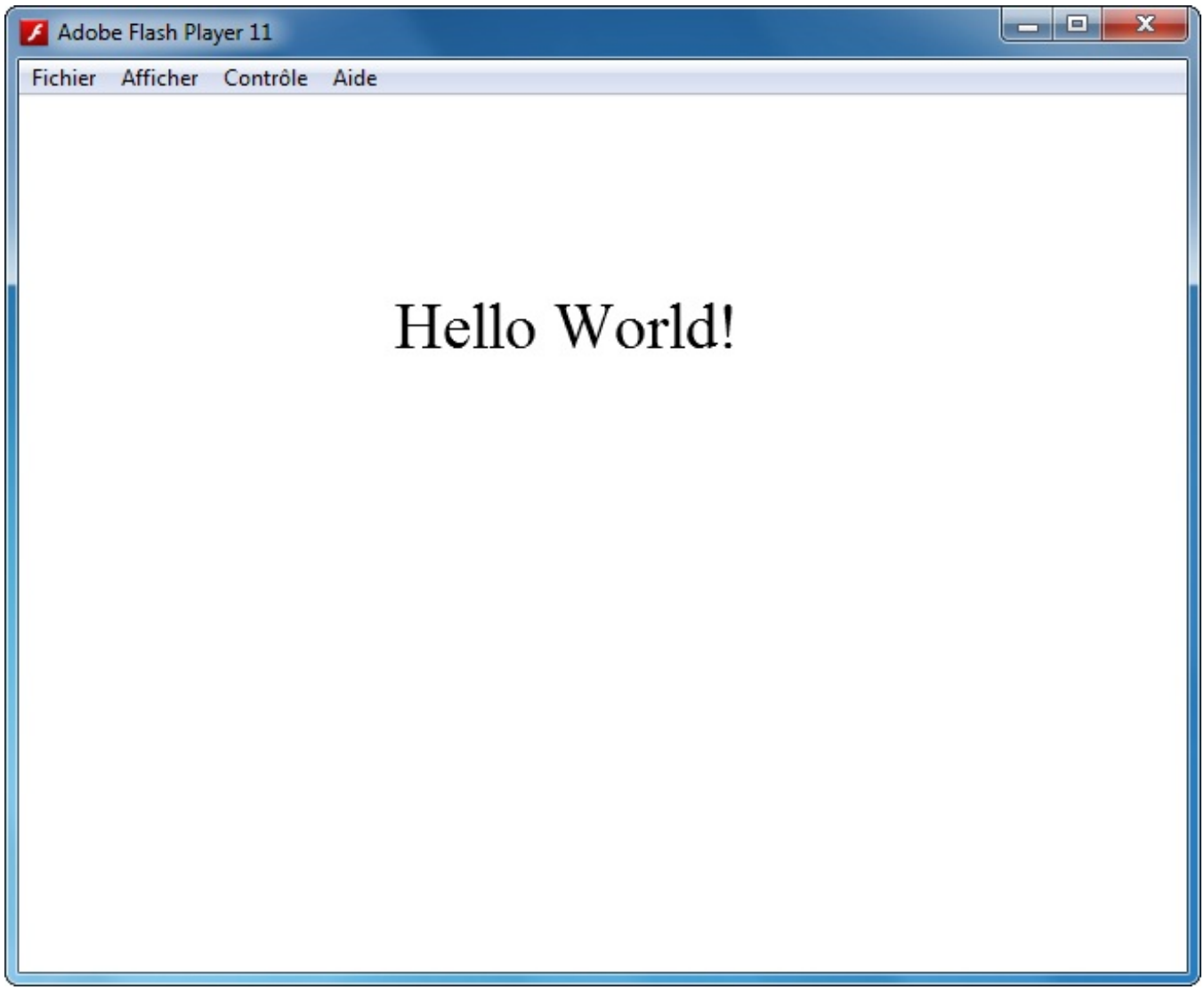
Position à (200, 100)

Je le trouve un peu trop petit, pas vous ? Allez, triplons sa taille :

Code : Actionscript

```
// Taille x3 !
monSuperTexte.scaleX = 3;
monSuperTexte.scaleY = 3;
```

Ce qui donne à l'écran la figure suivante.



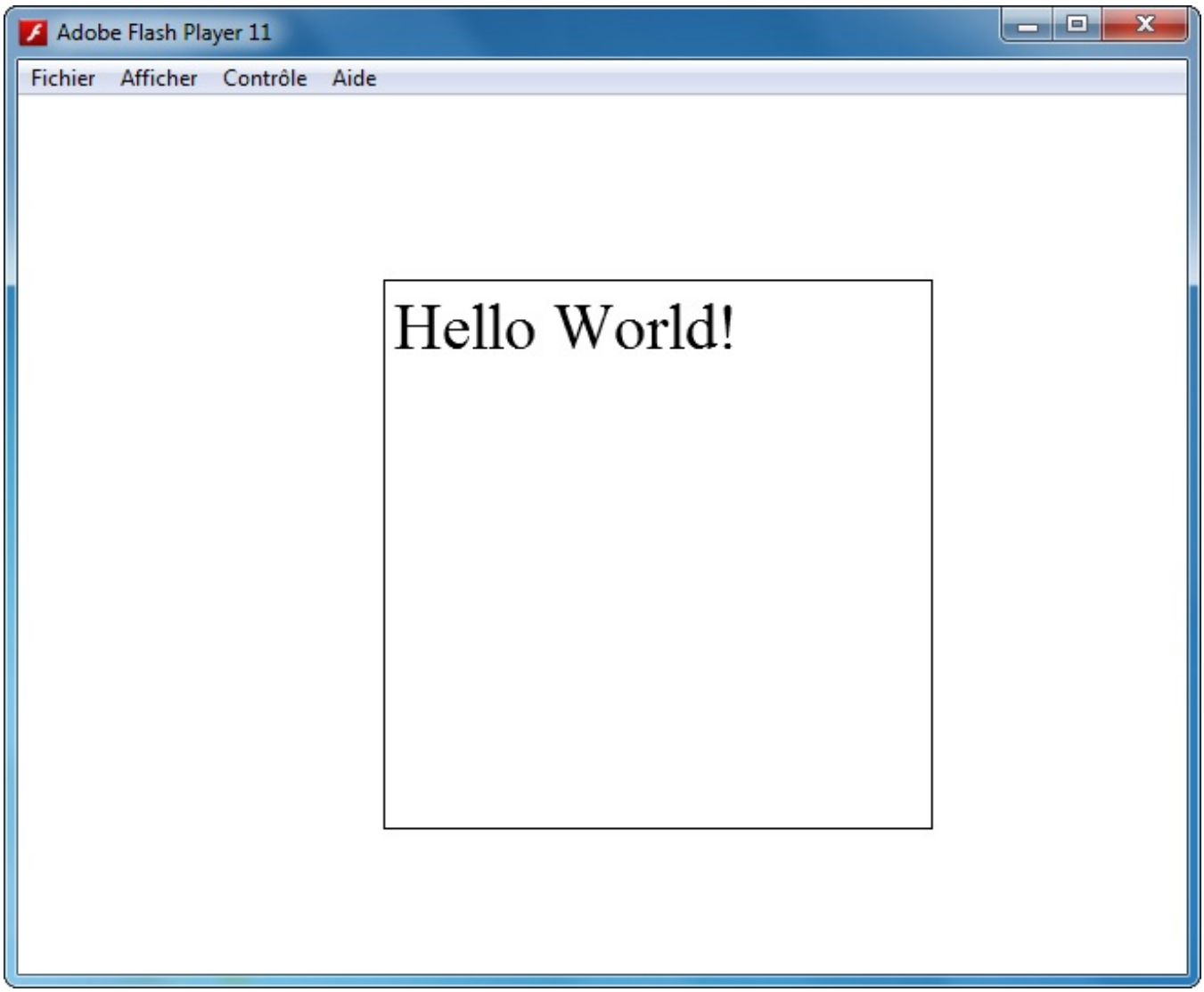
Notre champ de texte, quatre fois plus grand

Désormais, la taille de notre champ de texte est 300 pixels de largeur et 300 pixels de hauteur. Pour mieux le voir, nous pouvons remettre la bordure à notre champ de texte en décommentant cette ligne :

Code : Actionscript

```
monSuperTexte.border = true;
```

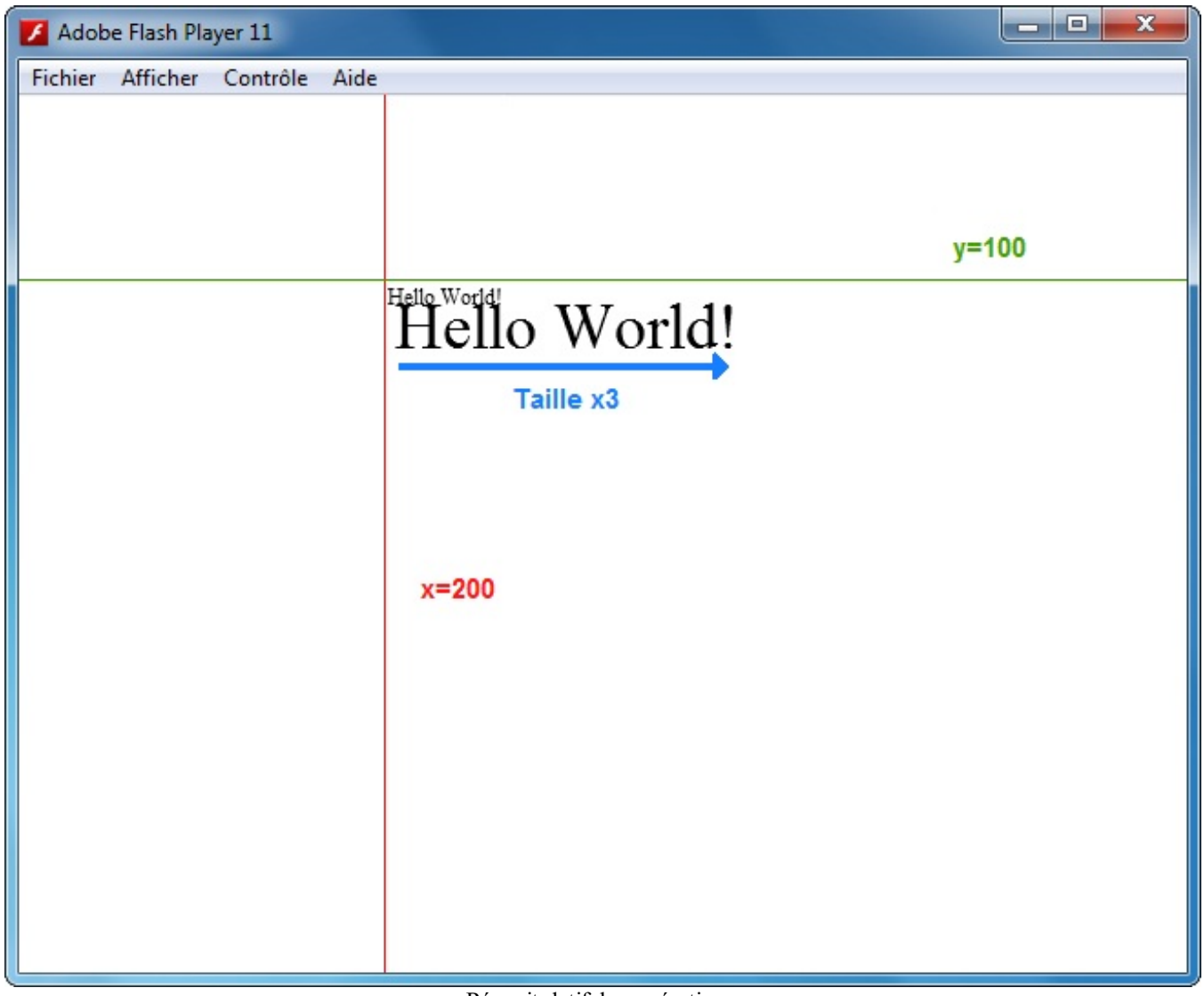
Ce qui donne à l'écran la figure suivante.



Le champ de texte agrandi, avec sa bordure

Pour récapituler (voir figure suivante), nous avons :

- créé une instance de la classe `TextField` ;
- défini le texte du champ de texte ;
- ajouté le champ de texte à l'affichage ;
- modifié sa position ;
- modifié son échelle de taille.



Récapitulatif des opérations

Il est impossible de faire tourner du texte ou de changer son opacité dans l'état actuel, car nous utilisons le système pour rendre les caractères, ce qui implique ces limitations. Si vous changez tout de même la rotation ou l'opacité du champ de texte, les caractères ne seront plus affichés. Pour contourner ce problème, il faut utiliser des **polices de caractères embarquées**, afin que les caractères soient rendus directement par le lecteur Flash. Nous apprendrons comment faire à la fin de ce chapitre. 

Sélection du texte

Curseur d'insertion

Tous les champs de texte disposent d'un curseur d'insertion, représenté par une barre verticale. Lorsque le champ de texte n'est pas éditable, on dit qu'il est **statique**. Dans ce cas, le curseur d'insertion est toujours invisible. Jusqu'ici, notre champ de texte est statique.

Définir une sélection

Il est possible de sélectionner une portion de texte directement à l'aide de la méthode `setSelection()` de la classe `TextField`, qui prend en paramètre la position du premier caractère à sélectionner et la position du caractère après le dernier caractère de la sélection. Par exemple, pour sélectionner les deux premières lettres, on écrira ceci :

Code : Actionscript

```
monSuperTexte.setSelection(0, 2);
```

Nous mettons 2 en deuxième paramètre, car il faut passer la position du caractère après le dernier caractère à sélectionner, c'est-à-dire la position du dernier caractère à sélectionner (le numéro 1) plus une.

Si vous testez le programme maintenant, vous ne verrez pas le texte sélectionné : c'est normal, car par défaut les champs de texte qui n'ont pas le focus, c'est-à-dire qui ne sont pas actifs, n'affichent pas la sélection du texte. Pour rendre un champ de texte actif, la manière la plus simple est de cliquer dessus.
Il existe cependant un moyen d'afficher tout de même la sélection, même si le champ de texte n'a pas le focus : on met l'attribut `alwaysShowSelection` à **`true`** :

Code : Actionscript

```
monSuperTexte.alwaysShowSelection = true;
```

Ce qui nous donne à l'écran la figure suivante.

Les deux premières lettres sont sélectionnées.

Accéder à la sélection

Il est possible de savoir à quelles positions le texte sélectionné débute et se termine. Nous disposons pour cela des attributs `selectionBeginIndex`, position du premier caractère sélectionné, et `selectionEndIndex`, position du dernier caractère sélectionné :

Code : Actionscript

```
trace("Sélection : " + monSuperTexte.selectionBeginIndex + " -> " +
monSuperTexte.selectionEndIndex); // Affiche 0 -> 2
```

Pour avoir le texte sélectionné, nous utiliserons la méthode `substring` de l'attribut `text:String` du champ de texte :

Code : Actionscript

```
trace("Texte sélectionné : " +
monSuperTexte.text.substring(monSuperTexte.selectionBeginIndex,
monSuperTexte.selectionEndIndex)); // Affiche : He
```

La valeur renvoyée par `selectionEndIndex` correspond à la position du caractère après le dernier caractère sélectionné (autrement dit, la position du dernier caractère sélectionné + 1). Mais cela tombe bien, car la méthode `substring` prend les caractères jusqu'à la deuxième position passée en paramètre moins une. Nous obtenons donc le bon texte.

Empêcher la sélection du texte

Par défaut, le texte contenu par un champ de texte peut être sélectionné par l'utilisateur, puis éventuellement copié. Le curseur se change alors en curseur de texte.
Ce comportement peut être gênant, par exemple, si vous utilisez un champ de texte pour créer un bouton. Pour le désactiver, il suffit de mettre l'attribut `selectable` de la classe `TextField` à **`false`** :

Code : Actionscript

```
monSuperTexte.selectable = false;
```

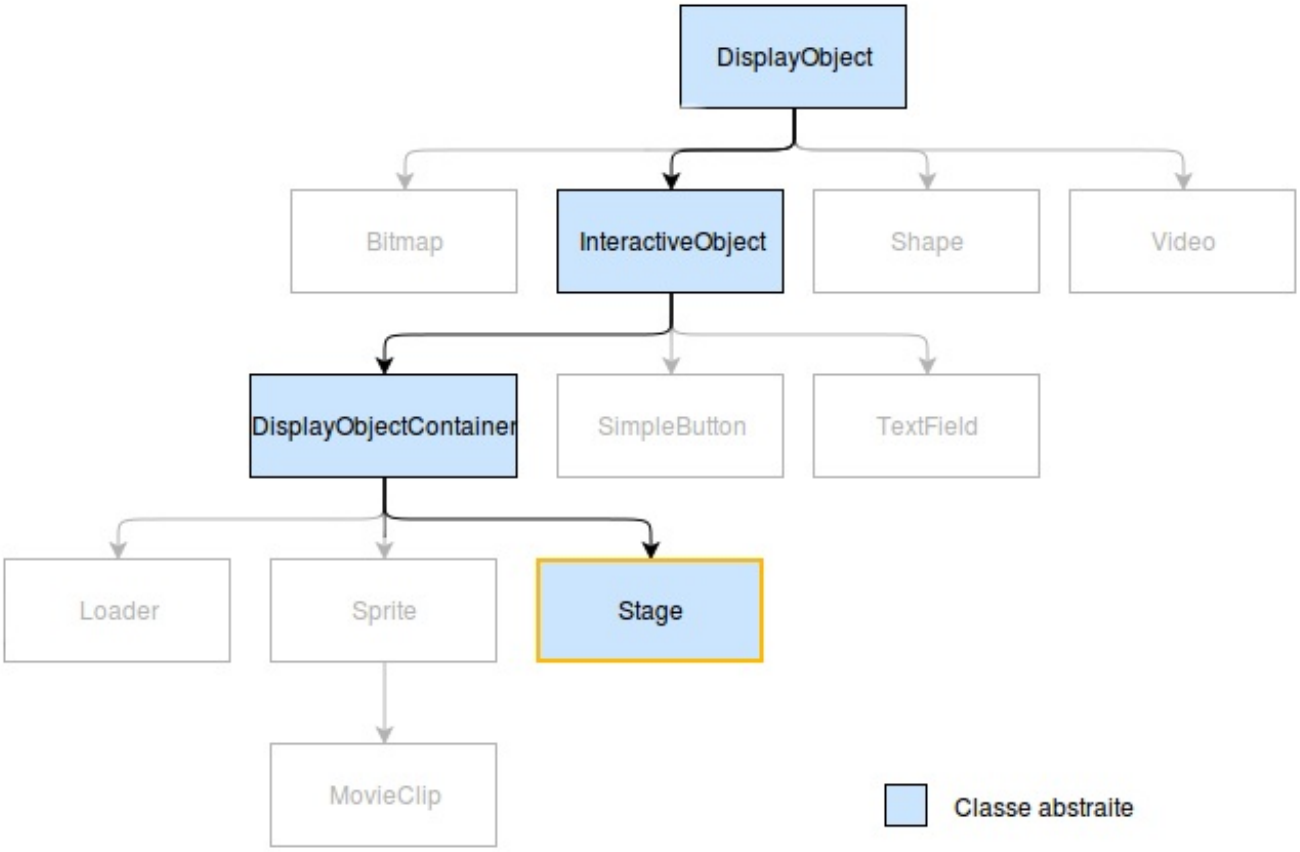
Centrer le champ de texte

Comme nous l'avons placé jusqu'ici, le texte c'est pas exactement au centre de la fenêtre. Pour remédier à cela, il nous faut connaître la taille exacte de la scène !

Un mot sur la scène principale

La classe `Stage`

Comme nous l'avoir vu précédemment, une instance de la classe `Stage` est automatiquement créée au lancement de notre programme : il s'agit de la scène principale. La scène est un objet d'affichage comme on peut le voir sur le schéma des classes d'affichage à la figure suivante.



La classe Stage parmis les autres classes d'affichage

La classe `Stage` est donc une sous-classe de la classe `DisplayObjectContainer` : elle hérite donc de toutes ses propriétés, et s'apparente à un conteneur comme nous en avons vu au chapitre précédent. Mais il y a tout de même quelques différences :

- on ne peut pas créer de scène principale nous-même, car la classe `Stage` est abstraite,
- la plupart des propriétés de la classe `DisplayObjectContainer` ne fonctionnent pas,
- la classe `Stage` dispose de nouvelles propriétés spécifiques.

Voici une liste des propriétés que nous avons déjà abordées et qui sont sans effet pour la classe `Stage` :

- alpha
- rotation
- scaleX
- scaleY

- visible
- x
- y

Inutile donc d'essayer de modifier ces attributs, cela n'aura aucun impact sur la scène principale.
Par contre, il est toujours possible d'utiliser les attributs de l'instance de notre classe `Main`, afin d'impacter l'ensemble de notre arbre d'affichage. Par exemple, si l'on veut rendre la scène invisible, nous pouvons mettre ce code dans notre classe `Main` :

Code : Actionscript

```
// Dans la classe Main
this.visible = false; // Toute la scène est invisible !

visible = false; // On peut omettre le mot-clé this
```

Il est possible d'accéder à la scène principale grâce à l'attribut `stage` disponible sur n'importe quel objet d'affichage dérivé de la classe `DisplayObject`, **à condition que cet objet soit présent dans l'arbre d'affichage**.

Code : Actionscript

```
// On ne peut accéder à la scène depuis notre objet voiture
seulement s'il est présent dans l'arbre d'affichage :
var voiture:Sprite = new Sprite();
trace(voiture.stage); // Affiche null
addChild(voiture);
trace(voiture.stage); // Affiche [object Stage]
```

Cela signifie que l'on peut accéder à la scène principale depuis notre classe Main comme ceci :

Code : Actionscript

```
// Dans la classe Main, dans la méthode init()
trace(this.stage); // Affiche [object Stage]

// Comme d'habitude, il est possible d'omettre le mot-clé this :
trace(stage); // Affiche [object Stage]
```

Le code par défaut proposé par FlashDevelop

Il est temps de vous expliquer pourquoi le code par défaut de notre classe Main contient plus de choses que pour les autres classes que nous avons faites, lorsque nous utilisons *FlashDevelop*. Comme nous l'avons vu précédemment, on ne peut accéder à la scène d'affichage que si notre objet est présent dans l'arbre d'affichage. Ainsi, le code par défaut de notre classe `Main` permet de vérifier si notre objet de la classe est bien présent dans l'arbre avant d'appeler la méthode `init()`. Si jamais ce n'était pas encore le cas (par exemple, si le lecteur Flash met un peut de temps à lancer le programme), nous aurions des problèmes pour utiliser les propriétés de la scène principale : en effet, l'attribut `stage` serait à **`null`** !
Reprenons le code par défaut de la classe `Main` et commentons un peu pour mieux comprendre :

Code : Actionscript

```
package
{
    // Classes nécessaires
    import flash.display.Sprite;
    import flash.events.Event;

    /**
* La classe de base sans modification.
```

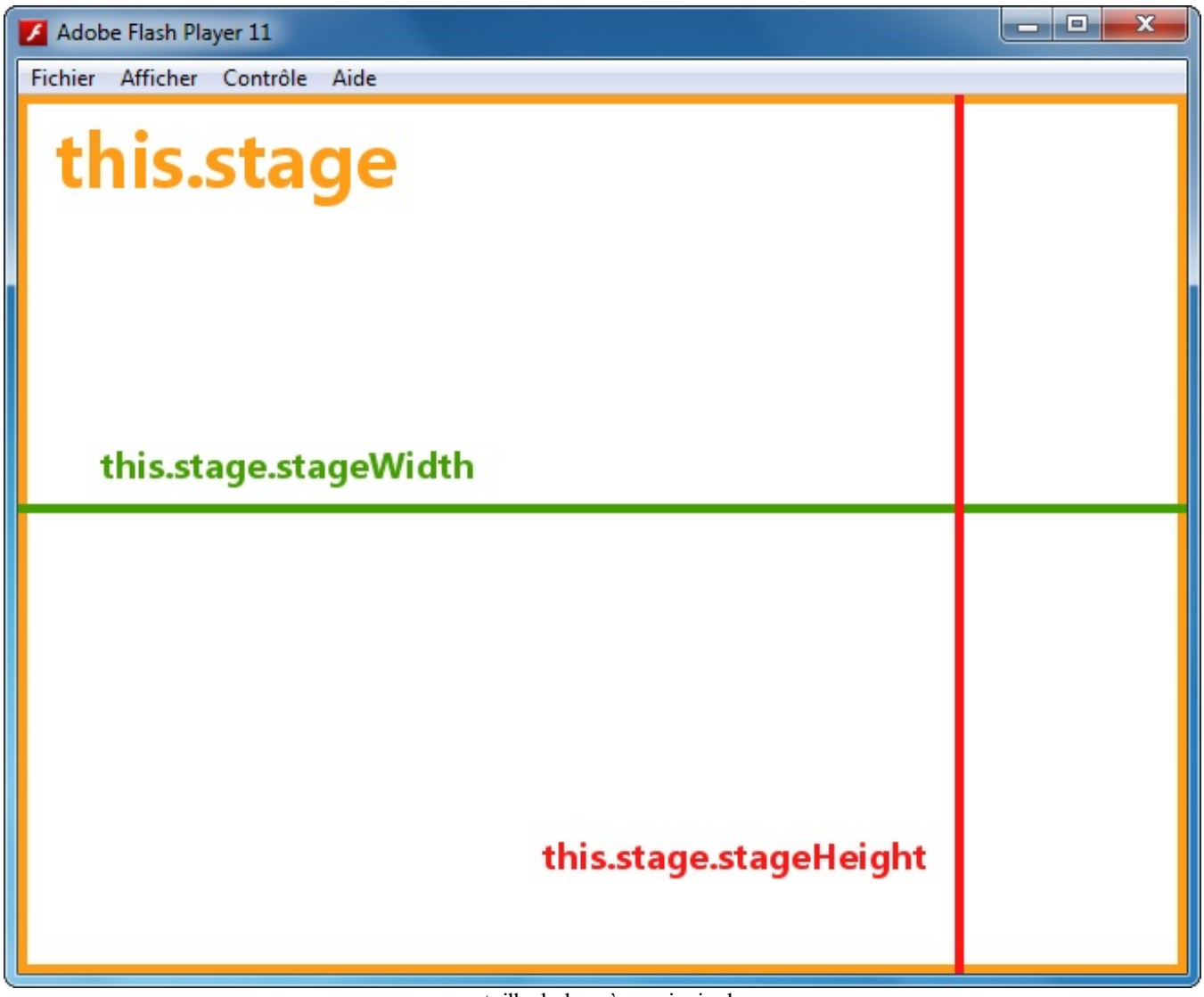
```
*/
    [SWF(width="640",height="480",backgroundColor="#ffffff")]

    public class Main extends Sprite
    {
        // Constructeur
        public function Main():void
        {
            if (stage) // Si l'attribut stage est défini (c'est-à-
dire, si il est différent de null)
                init(); // Alors, on appelle la méthode init() tout
de suite
            else // Sinon
                addEventListener(Event.ADDED_TO_STAGE, init); // On
attend que notre instance de la classe Main soit ajoutée à l'arbre
de l'affichage
        }

        private function init(e:Event = null):void
        {
            removeEventListener(Event.ADDED_TO_STAGE, init); // On
arrête d'attendre

            // Nous pouvons commencer à coder ici en tout sérénité
:)
        }
    }

}
```

Le code qui permet d'attendre que l'objet soit ajouté à l'arbre d'affichage utilise la notion d'**événement**, notion que nous aborderons dans la partie suivante.

La taille de la scène

Revenons à nos moutons et intéressons-nous à la taille de la fenêtre de notre programme. On peut y accéder grâce à deux attributs de la classe `Stage` (voir figure suivante) :

- `stageWidth` pour connaître la largeur de la scène,
- `stageHeight` pour la hauteur.



La taille de la scène principale

Par exemple, nous pouvons écrire ceci dans notre classe Main pour afficher la taille de la scène principale :

Code : Actionscript

```
trace("La scène principale a pour dimensions " + stage.stageWidth +
"x" + stage.stageHeight);
```

La console affichera alors :

Code : Console

```
La scène principale a pour dimensions 640x480
```

Cela correspond à la configuration de notre programme qui se trouve à cette ligne :

Code : Actionscript

```
[SWF(width="640",height="480",backgroundColor="#ffffff")]
```

Il ne faut pas confondre les attributs `stageWidth` et `stageHeight` qui donnent les dimensions de la scène principale, avec les attributs `width` et `height` qui donnent la taille du contenu de la scène !

En effet, nous n'obtenons pas le même résultat dans les deux cas. Pour le vérifiez, ajoutons ce code :

Code : Actionscript

```
trace("Le contenu de la scène principale a pour taille " +
stage.width + "x" + stage.height);
```

Ensuite, la console affiche ceci :

Code : Console

```
La scène principale a pour dimensions 640x480
Le contenu de la scène principale a pour taille 300x300
```

Comme le montre la figure suivante, la taille du contenu de la scène est souvent différente des dimensions de la scène :

La taille du contenu de la scène par rapport aux dimensions de la scène

Adapter la taille du champ de texte au texte

Pour centrer le champ de texte, il serait intéressant que celui-ci fasse la même taille que celle du texte à l'intérieur. Pour l'instant, notre champ de texte fait 300 pixels de largeur et 300 pixels de hauteur, ce qui est beaucoup trop. Nous pouvons dire au champ de texte d'adapter sa taille grâce à son attribut `autoSize` :

Code : Actionscript

```
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;
```

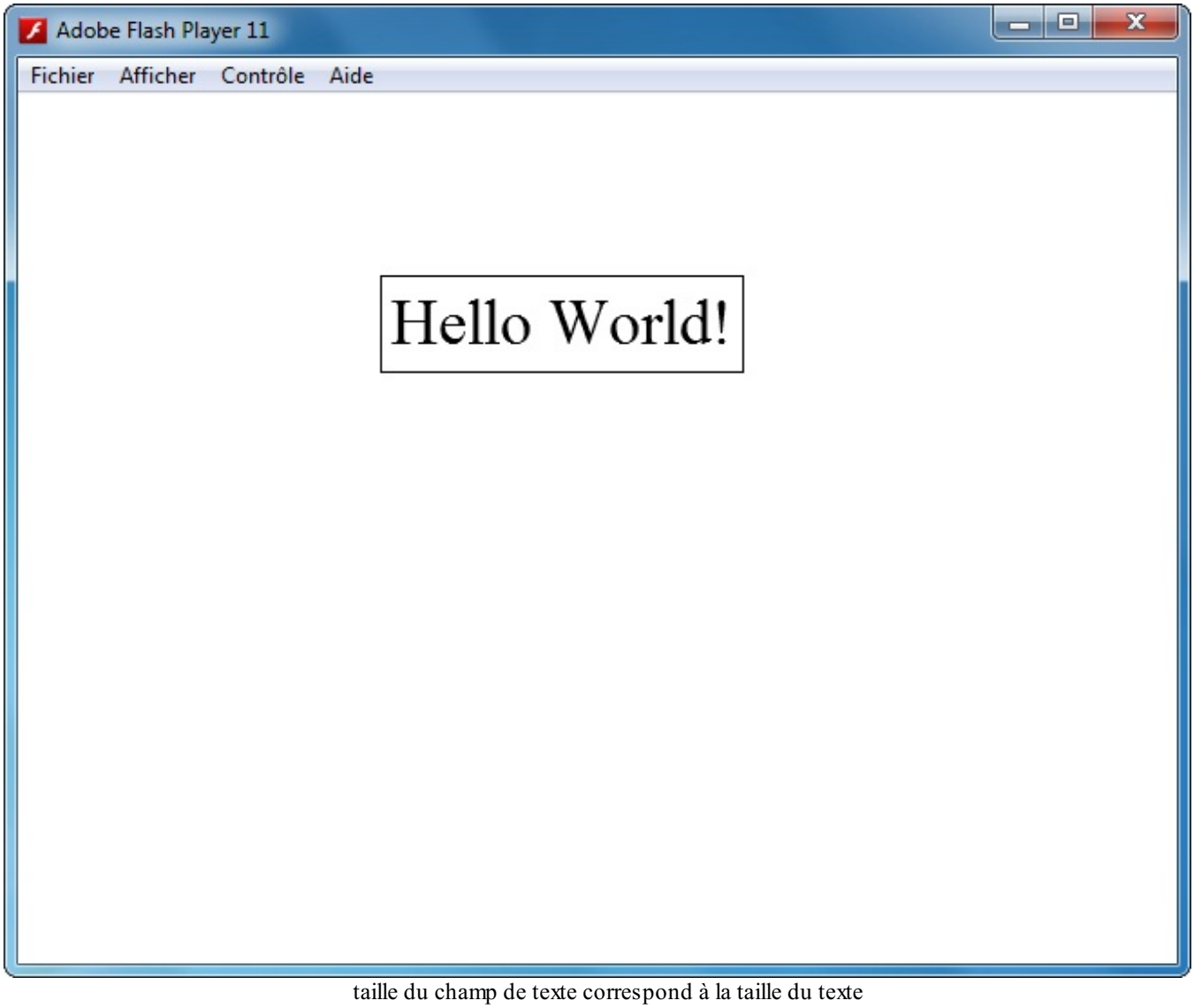
Sans oublier d'importer la classe TextFieldAutoSize (qui contient les valeurs autorisées pour l'attribut `autoSize`) :

Code : Actionscript

```
import flash.text.TextFieldAutoSize;
```

Petite astuce si vous programmez avec `FlashDevelop` : tapez **`new`** `Text` et l'auto-complétion vous proposera la classe. Sélectionnez-la et elle sera automatiquement importée. Ensuite, n'oubliez pas de supprimer le mot-clé **`new`** dont nous n'avons pas besoin.

Désormais, nous avons à l'écran la figure suivante.



La taille du champ de texte correspond à la taille du texte

Nous pouvons afficher la taille du champ de texte pour être sûr de nous :

Code : Actionscript

```
trace("Taille : " + monSuperTexte.width + "x" +
monSuperTexte.height); // Affiche 199.95x52.95
```

Modifier la position du champ de texte

Il est temps de centrer notre notre champ de texte ! Nous allons procéder en trois étapes pour bien détailler l'opération.

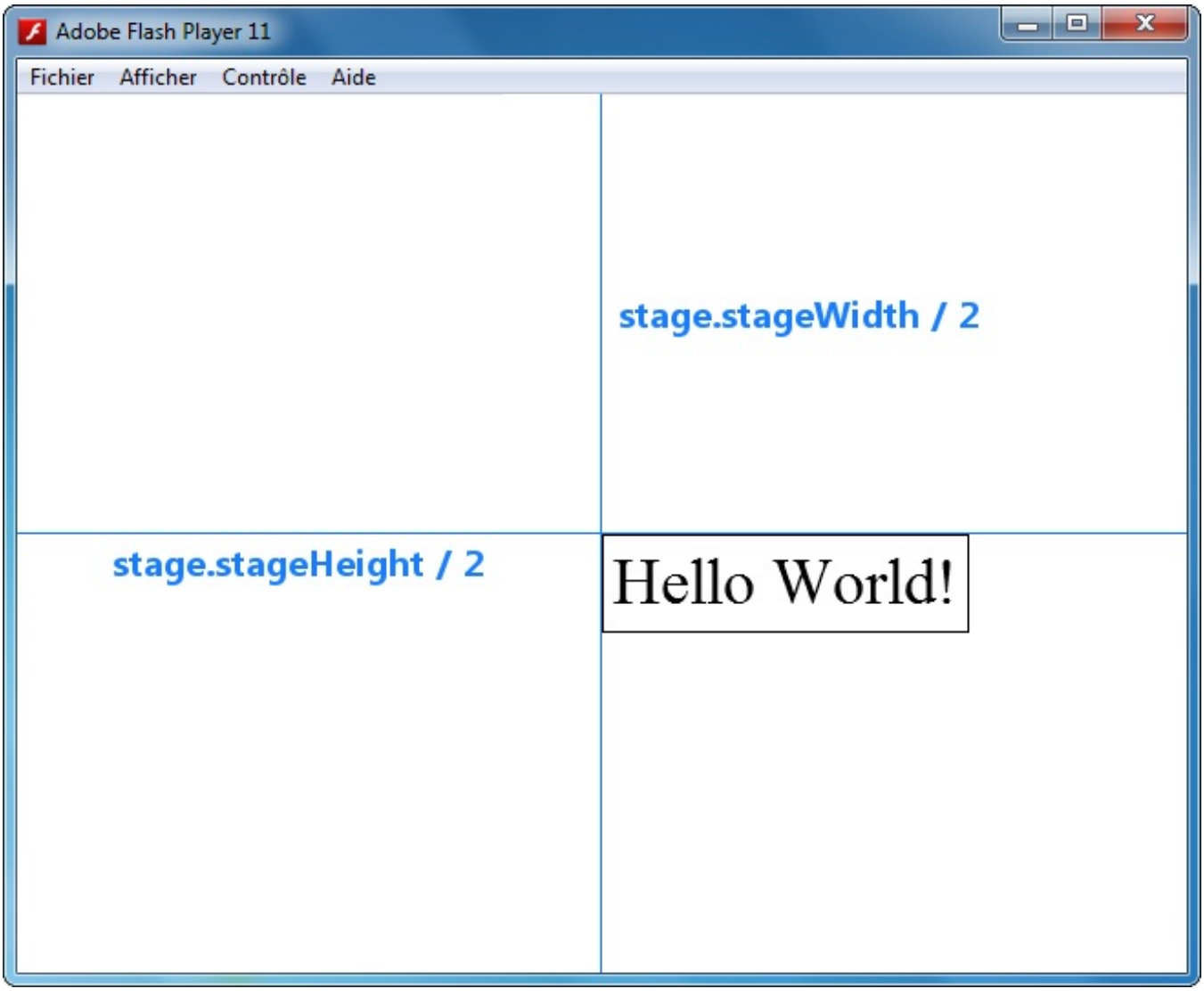
Etape 1

Commençons par placer l'origine de notre champ de texte au centre de l'écran. Pour cela, il suffit d'utiliser les attributs `stageWidth` et `stageHeight` que nous venons de voir :

Code : Actionscript

```
monSuperTexte.x = stage.stageWidth / 2;
monSuperTexte.y = stage.stageHeight / 2;
```

Ce qui nous donne à l'écran la figure suivante.



L'origine du champ de texte est centrée

Notre champ de texte n'est pas encore entièrement centré, mais nous y sommes presque !

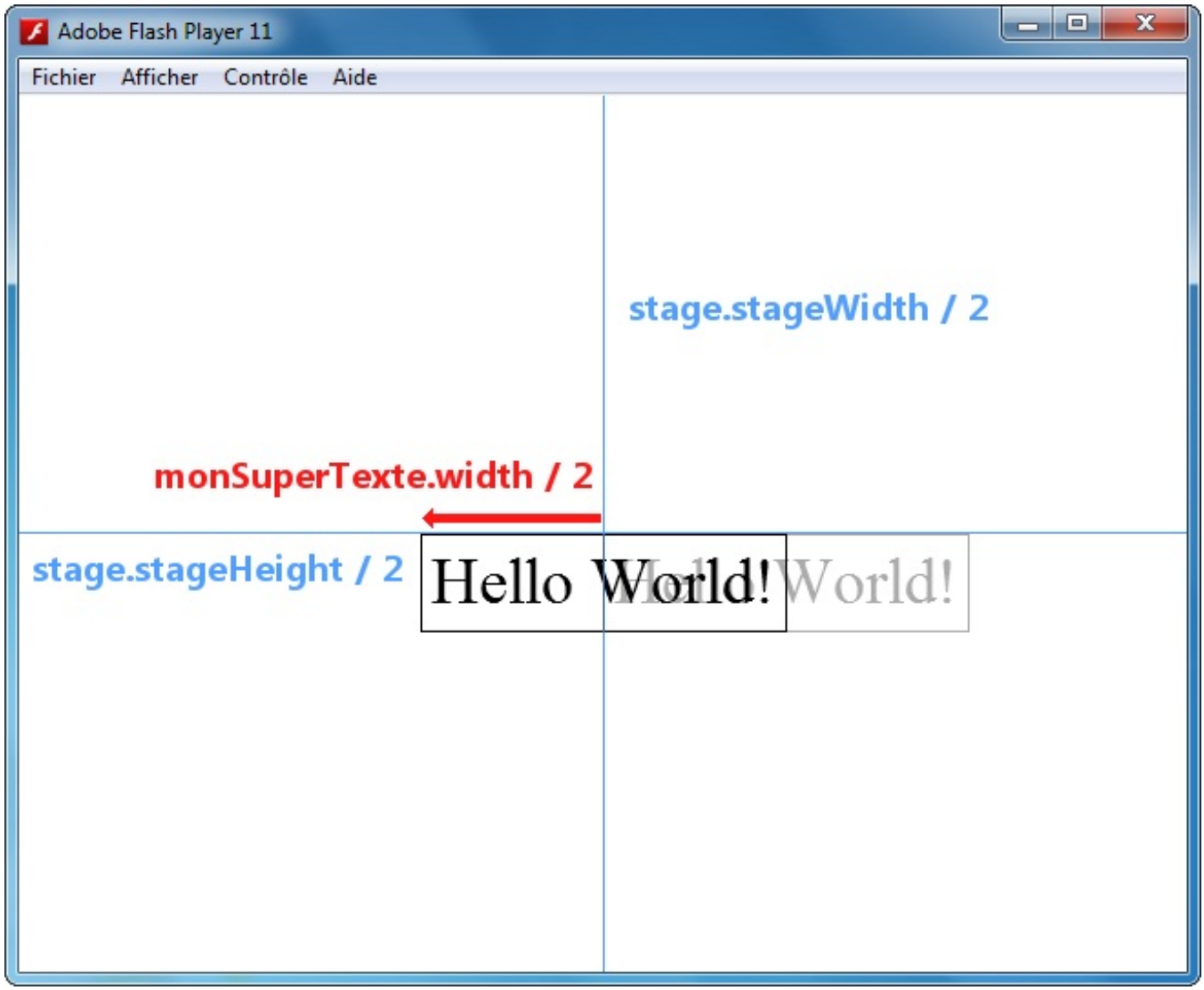
Etape 2

Nous allons maintenant retrancher la moitié de la largeur de notre champ de texte à sa position `x` pour centrer notre champ de texte horizontalement :

Code : Actionscript

```
monSuperTexte.x -= monSuperTexte.width / 2;
```

Ce qui nous donne à l'écran la figure suivante.



Notre champ de texte, centré horizontalement

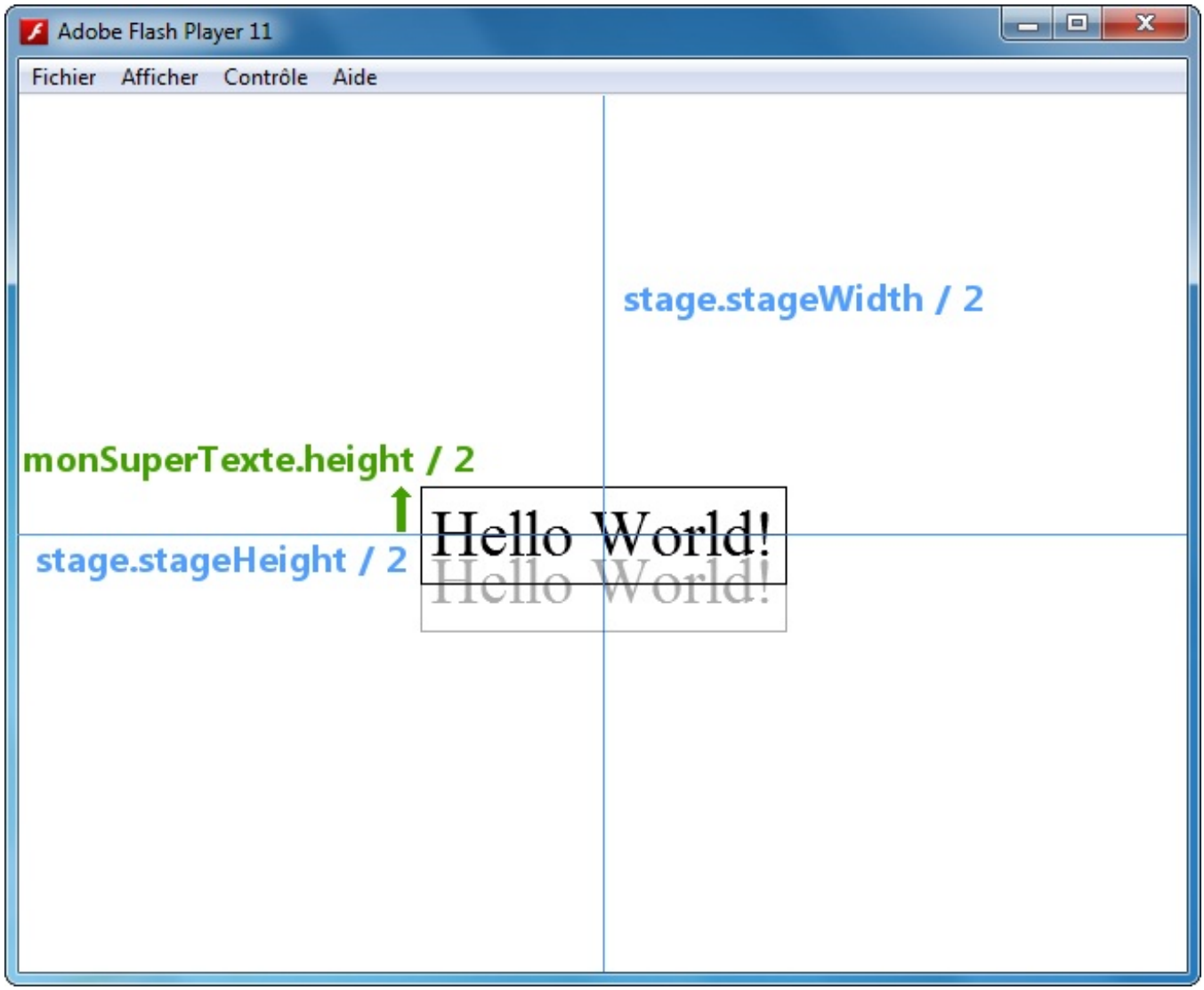
Etape 3

Et pour finir, nous répétons la même chose, mais verticalement : nous retranchons la moitié de la hauteur de notre champ de texte à sa position `y` :

Code : Actionscript

```
monSuperTexte.y -= monSuperTexte.height / 2;
```

Ce qui nous donne la figure suivante.



Notre champ de texte, au centre de la scène !

En résumé

L'opération s'est donc déroulée en trois étapes :

- on centre l'origine de l'objet en utilisant `stage.stageWidth` et `stage.stageHeight`,
- on retranche la moitié de sa largeur à sa position horizontale `x`,
- on retranche la moitié de sa hauteur à sa position verticale `y`.

Et voici le code complet concernant notre champ de texte :

Code : Actionscript

```
// Créons notre champ de texte
var monSuperTexte:TextField = new TextField();
monSuperTexte.text = 'Hello World!';
addChild(monSuperTexte);

//Multiplions sa taille par quatre
monSuperTexte.scaleX = 4;
monSuperTexte.scaleY = 4;
```

```
// Activons la bordure
monSuperTexte.border = true;

// Adaptons la taille du champ de texte au texte
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;

// Centrons l'origine
monSuperTexte.x = stage.stageWidth / 2;
monSuperTexte.y = stage.stageHeight / 2;

// Retranchons la moitié de sa taille à sa position
monSuperTexte.x -= monSuperTexte.width / 2;
monSuperTexte.y -= monSuperTexte.height / 2;
```

Nous pouvons compacter la dernière partie de notre code, qui permet de centrer le champ de texte :

Code : Actionscript

```
// Centrons le champ de texte
monSuperTexte.x = stage.stageWidth / 2 - monSuperTexte.width / 2;
monSuperTexte.y = stage.stageHeight / 2 - monSuperTexte.height / 2;
```

Nous pouvons encore simplifier en factorisant :

Code : Actionscript

```
// Centrons le champ de texte
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;
```

Cette méthode pour aligner au centre de la scène fonctionne bien évidement avec tous les objets d'affichage, pas seulement les champs de texte. 

Pour l'instant, il nous est impossible de faire tourner notre champ de texte, car il utilise une police de votre système pour le rendu. Pour contourner ce problème, il nous faut embarquer notre propre police de caractère, comme nous le verrons dans un prochain chapitre.

La mise en forme

Formatons notre champ de texte

Mise en forme par défaut

Formater du texte est possible grâce à la classe `TextFormat` (dont la documentation est disponible ici). Tout objet de classe `TextField` dispose d'un attribut `defaultTextFormat` de type `TextFormat`. Cet attribut permet de modifier la mise en forme par défaut du texte contenu dans le champ de texte, cela signifie qu'un texte qui n'a pas de mise en forme spécifique recevra automatiquement cette mise en forme par défaut.

La mise en forme par défaut doit être appliquée avant de définir le texte.

Pour modifier la mise en forme par défaut de notre champ de texte, nous procéderons ainsi :

Code : Actionscript

```
// Création de l'instance du champ de texte
var monSuperTexte:TextField = new TextField();

// Création de l'instance de la classe TextFormat
```

```
var format:TextFormat = new TextFormat();

// Modifications du format ici

// Application du format au champ de texte
monSuperTexte.defaultTextFormat = format;

// Texte à afficher - après la mise en forme par défaut
monSuperTexte.text = "Hello World!";

// Ajout du champ de texte à l'affichage
addChild(monSuperTexte);
```

La mise en forme par défaut n'est mise à jour que si nous affectons un objet de classe `TextFormat` à l'attribut `defaultTextFormat`. Si nous modifions directement les attributs de `defaultTextFormat`, il ne se passera rien. Ne pas boulier qu'il faut que le texte soit défini après avoir mis à jour la mise en forme par défaut.

L'heure du changement est arrivée

Nous allons dans un premier temps modifier la taille, la police de caractère et la couleur de notre texte. En se renseignant dans la documentation, nous pouvons noter les différents attributs correspondants :

- `size:Object` (entier) pour la taille du texte en pixels,
- `font:String` (chaîne de caractère) pour le nom de la police de caractère à utiliser,
- `color:Object` (entier positif) pour la couleur du texte.

Avant de commencer, rendons sa taille normale à notre champ de texte en supprimant (ou commentant) les lignes qui modifient son échelle :

Code : Actionscript

```
// monSuperTexte.scaleX = 3;
// monSuperTexte.scaleY = 3;
```

Exerçons-nous à modifier la mise en forme de notre champ de texte : notre texte doit être de taille `42`, utilisant la police de caractère `Arial` et colorié en bleu (dont le code hexadécimal est `0x0000ff`).

Code : Actionscript

```
// Création de l'instance du champ de texte
var monSuperTexte:TextField = new TextField();

// Création de l'instance de la classe TextFormat
var format:TextFormat = new TextFormat();

// Mise en forme
format.size = 42;
format.font = "Arial";
format.color = 0x0000ff;

// Application du format au champ de texte
monSuperTexte.defaultTextFormat = format;

// Texte
monSuperTexte.text = "Hello World!";

// Bordure
monSuperTexte.border = true;

// Taille adaptative
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;
```

```
// Centrons le champ de texte
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;

// Ajout du champ de texte à l'affichage
addChild(monSuperTexte);
```

Ce qui nous donne la figure suivante.

Le texte bleu, en Arial et en taille 42

Vous remarquerez qu'il est possible de positionner l'objet d'affichage avant qu'il ne soit ajouté à l'arbre d'affichage.

Si vous vous trompez en écrivant le nom de la police, ou que celle-ci n'est pas installée sur le système de l'utilisateur, la police par défaut sera utilisée à la place.

Pourquoi certains de ces attributs ont pour type `Object` au lieu de `int` ou `uint` ?

C'est très simple : les attributs de la classe `TextFormat` ont pour valeur par défaut **`null`**, pour signifier qu'il faut utiliser les valeurs par défaut de mise en forme (que l'on peut trouver dans la documentation, en haut de la page). Or, les attributs de type `int` ou `uint` ne peuvent pas valoir **`null`**, donc on est obligé d'utiliser le type `Object` (type indéfini, dont les variables peuvent valoir **`null`**). On aurait aussi pu utiliser le type `*`, équivalent au type `Object`.

Le constructeur de la classe `TextFormat` permet de définir directement certains attributs. Ainsi, nous pourrions simplifier notre code en utilisant le constructeur de la classe `TextFormat` ainsi que nous l'indique la documentation :

Code : Actionscript

```
TextFormat(font:String = null, size:Object = null, color:Object =
null, bold:Object = null, italic:Object = null, underline:Object =
null,
    url:String = null, target:String = null, align:String = null,
leftMargin:Object = null, rightMargin:Object = null, indent:Object =
null, leading:Object = null)
```

Nous pouvons donc écrire :

Code : Actionscript

```
// Création de l'instance de la classe TextFormat
var format:TextFormat = new TextFormat("Arial", 42, 0x0000ff);

// Application du format au champ de texte
monSuperTexte.defaultTextFormat = format;
```

Nous pouvons également mettre le texte en gras, italique, ou souligné, grâce à ces attributs de la classe `TextFormat` :

- `bold:Object` (booléen) pour mettre le texte en gras,
- `italic:Object` (booléen) pour le mettre en italique,
- `underline:Object` (booléen) pour le souligner.

Notre code pour la mise en forme ressemble maintenant à ceci :

Code : Actionscript

```
// Mise en forme
format.size = 42;
format.font = "Arial";
format.color = 0x0000ff;
format.bold = true;
format.italic = true;
format.underline = true;
```

Nous pouvons également utiliser le constructeur :

Code : Actionscript

```
// Création de l'instance de la classe TextFormat
var format:TextFormat = new TextFormat("Arial", 42, 0x0000ff, true,
true, true);

// Application du format au champ de texte
monSuperTexte.defaultTextFormat = format;
```

Ce qui nous donne la figure suivante.

Le texte en gras, italique et souligné

Nous avons seulement vu les principaux attributs, mais rien ne vous empêche de jeter un coup d'œil à la documentation pour voir tout ce qu'il est possible de faire.

Modifier la mise en forme par défaut

Une fois la mise en forme par défaut définie, le texte que nous ajoutons à notre champ de texte prend par défaut cette mise en forme. Mais ensuite, vous aurez sûrement envie de modifier cette mise en forme par défaut. Avec un peu d'astuce, c'est possible ! Il suffit d'affecter de nouveau notre objet de classe `TextFormat` à l'attribut `defaultTextFormat` de notre champ de texte, puis réaffecter son texte :

Code : Actionscript

```
// Création de l'instance du champ de texte
var monSuperTexte:TextField = new TextField();

// Création de l'instance de la classe TextFormat
var format:TextFormat = new TextFormat();

// Mise en forme
format.size = 42;
format.font = "Arial";
format.color = 0x0000ff;
format.bold = true;
format.italic = true;
format.underline = true;

// Application du format au champ de texte
monSuperTexte.defaultTextFormat = format;

// Texte
```

```
monSuperTexte.text = "Hello World!";

// Bordure
monSuperTexte.border = true;

// Taille adaptative
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;

// Centrons le champ de texte
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;

// Ajout du champ de texte à l'affichage
addChild(monSuperTexte);

// ...

// Changeons en cours de route pour du rouge
format.color = 0xff0000;

// Mise à jour du format par défaut
monSuperTexte.defaultTextFormat = format;

// Il faut redéfinir le texte pour appliquer la nouvelle mise en
forme par défaut
monSuperTexte.text = monSuperTexte.text;
```

Ce qui affiche la figure suivante.

Nous avons troqué le bleu pour du rouge

Mise en forme spécifique

Il est possible d'écraser la mise en forme par défaut pour tout ou une partie du texte, grâce à la méthode `setTextFormat()` de la classe `TextField`, qui prend en paramètre un objet de la classe `TextFormat`, la position du premier caractère et la position du caractère après le dernier caractère du texte dont on souhaite modifier la mise en forme. Ces deux positions commencent par zéro, comme pour les tableaux.

Rappel : si vous ne souhaitez pas changer des attributs dans la mise en forme spécifique, il faut que ces attributs valent **`null`**. Ainsi ils n'aurons pas d'effet sur le texte.
Ainsi, il est préférable de créer une nouvelle instance de la classe `TextFormat`, pour être sûr que tous les attributs sont à **`null`**.

Changeons la taille de la première lettre pour `72`. La première lettre est à la position `0`, et la lettre après elle est à la position `1` :

Code : Actionscript

```
// Nouvelle instance de la classe TextFormat (on peut réutiliser le
même pointeur "format")
format = new TextFormat();

format.size = 72;

// Application du nouveau format à la première lettre
monSuperTexte.setTextFormat(format, 0, 1);
```

Ce qui nous donne la figure suivante.

La première lettre est en taille 72

Comme vous pouver le remarquer, notre lettre est coupée en haut à droite ; c'est un effet de bord de l'italique. Nous pouvons remédier à cela grâce à l'attribut `letterSpacing` qui spécifie la largeur de l'espace entre les lettres. Mettons-la à 3 pixels :

Code : Actionscript

```
// Nouvelle instance de la classe TextFormat (on peut réutiliser le
même pointeur "format")
format = new TextFormat();

format.size = 72;
format.letterSpacing = 3;

// Application du nouveau format à la première lettre
monSuperTexte.setTextFormat(format, 0, 1);
```

Ainsi, il y aura 3 pixel d'espace entre la première lettre et la deuxième (voir figure suivante).

Le défaut est corrigé !

Essayons de colorier le mot "World" en bleu et de lui enlever l'italique !

Code : Actionscript

```
// Je stocke le mot que je veux colorier
var motBleu:String = "World";

// Je cherche les positions adéquates pour la méthode setTextFormat
// Position de début, grâce à la méthode indexOf de la classe
String
var positionDebut:int = monSuperTexte.text.indexOf(motBleu);
// Position de fin, en ajoutant la longueur du mot
var positionFin:int = positionDebut + motBleu.length;

// Je créé le format
format = new TextFormat();
format.color = 0x0000ff;
format.italic = false;

// J'applique le format aux positions que nous avons calculées
monSuperTexte.setTextFormat(format, positionDebut, positionFin);
```

Ce qui nous donne la figure suivante.

Le mot World en bleu sans l'italique

Gestion du multi-ligne

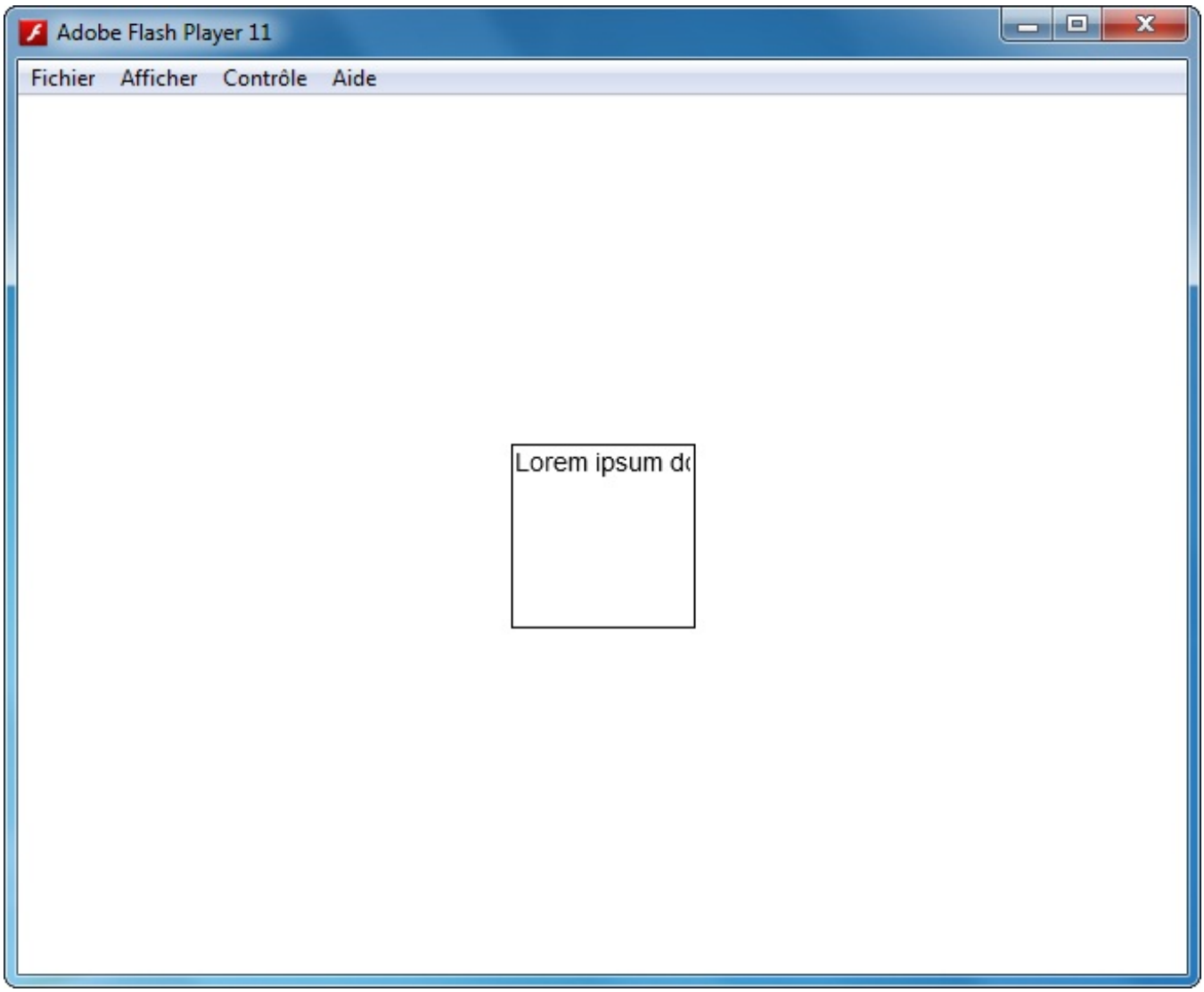
Champ de texte à taille fixe

Nous allons maintenant découvrir comment gérer un champ de texte multi-ligne. Tout d'abord, il nous faut un texte plus long que "Hello World!". Voici un Lorem Ipsum (texte latin de remplissage) pour tester :

Code : Actionscript

```
var monSuperTexte:TextField = new TextField();

var format:TextFormat = new TextFormat("Arial", 14, 0x000000);
monSuperTexte.defaultTextFormat = format;

// Texte suffisamment long
monSuperTexte.text = "Lorem ipsum dolor sit amet, consectetur
adipiscing elit. In pharetra magna imperdiet elit pretium a
malesuada nisl pellentesque.";

monSuperTexte.border = true;

addChild(monSuperTexte);

// Centre
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;
```

Pour l'instant, cela donne la figure suivante.



Le texte sort du champ de texte

Nous aimerions bien que le texte revienne automatiquement à la ligne. Pour cela, il faut mettre l'attribut `multiline` de la classe `TextField` à **`true`** pour activer les lignes multiples, et l'attribut `wordWrap` à **`true`** également, pour activer le retour à la ligne automatique :

Code : Actionscript

```
monSuperTexte.multiline = true;
monSuperTexte.wordWrap = true;
```

Ce qui donne à l'écran la figure suivante.

Retour à la ligne automatique

Le texte dépasse désormais en bas du champ de texte. Toutefois, par défaut, on peut défiler dans le champ de texte avec la molette de la souris. Si vous voulez désactiver ce comportement, il faut mettre l'attribut `mouseWheelEnabled` de la classe `TextField` à **`false`** :

Code : Actionscript

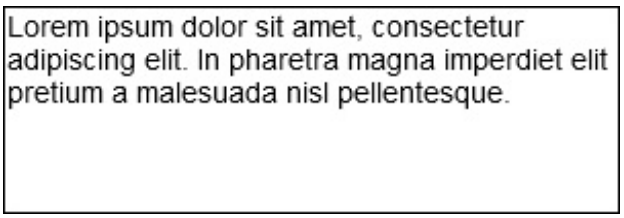
```
monSuperTexte.mouseWheelEnabled = false;
```

Le champ de texte est peut-être un peu trop étroit, augmentons sa largeur (avant le code qui le centre) :

Code : Actionscript

```
// Modifions la largeur du champ de texte
monSuperTexte.width = 300;

// Centre
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;
```

Ce qui nous donne la figure suivante.



Largeur de 300 pixels

Aligner le texte

Il est possible de changer l'alignement du texte grâce à l'attribut `align` de la classe `TextFormat`. Cet attribut accepte les valeurs constantes définies dans la classe `TextFormatAlign`.
Par exemple, pour que notre texte soit aligné au centre :

Code : Actionscript

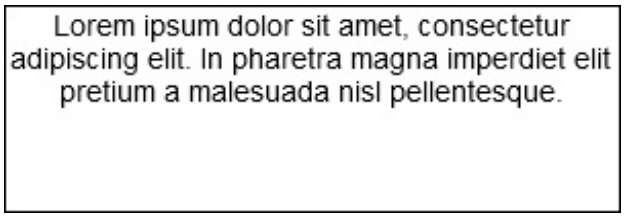
```
var format:TextFormat = new TextFormat("Arial", 14, 0x000000);
format.align = TextFormatAlign.CENTER; // Texte aligné au centre
monSuperTexte.defaultTextFormat = format;
```

Sans oublier de bien importer la classe `TextFormatAlign` au début du *package* :

Code : Actionscript

```
import flash.text.TextFormatAlign;
```

Cela nous donne la figure suivante.

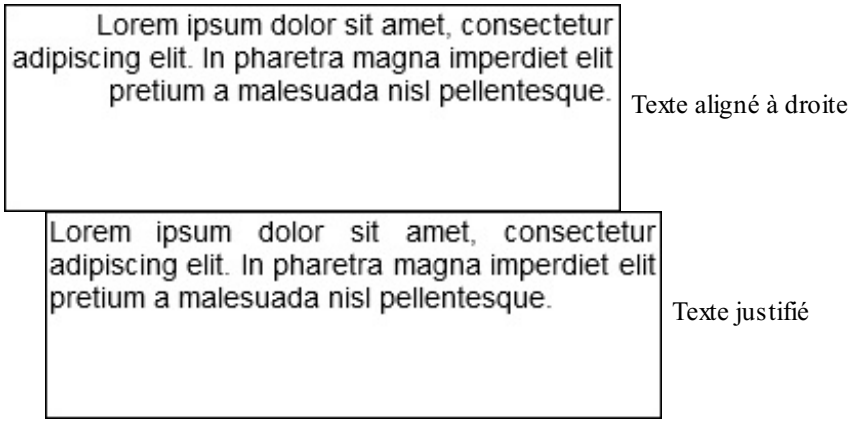


Texte aligné au centre

Nous pouvons également centrer le texte à droite ou le justifier (voir figure suivante) :

Code : Actionscript

```
// Texte aligné à droite
format.align = TextFormatAlign.RIGHT;

// Texte justifié
format.align = TextFormatAlign.JUSTIFY;
```



Un texte justifié est un texte dont les mots sont espacés de telle sorte que les lignes de chaque paragraphes remplissent toutes la largeur disponibles, sauf les dernières lignes. Par exemple, cet "alignement" est souvent utilisé dans les journaux.

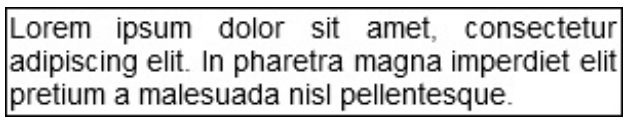
Champ de texte à hauteur dynamique

Il est possible d'adapter la hauteur du champ de texte à son texte s'il est multi-ligne, de la même manière que nous avions adapté la taille de notre champ de texte contenant le texte
`"Hello World!"` au chapitre précédent : avec l'attribut `autoSize` de la classe `TextField` :

Code : Actionscript

```
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;
```

Ce qui nous donne la figure suivante.



La hauteur est automatiquement adaptée au texte

Contrairement à ce que nous avons vu au chapitre précédent, seule la hauteur est adaptée au texte; la largeur est fixe et c'est vous qui la définissez. Ceci est uniquement valable si le champ de texte est multi-ligne, quand l'attribut `multiline` du champ de texte est à **`true`**.

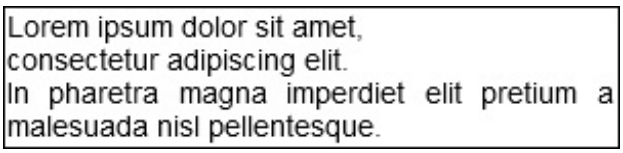
Retour à la ligne

Il est possible d'effectuer des retours à la ligne en écrivant `\n` dans le texte :

Code : Actionscript

```
monSuperTexte.text = "Lorem ipsum dolor sit amet,\nconsectetur
adipiscing elit.\nIn pharetra magna imperdiet elit pretium a
malesuada nisl pellentesque.";
```

Ce qui nous donne la figure suivante.



Deux retours à la ligne

En HTML dans le texte

Introduction

Le langage HTML

Ce langage est utilisé sur les sites web pour structurer le texte, ajouter des paragraphes, des images, des liens hyper-texte... Le langage est un dérivé du langage XML, ce qui signifie que sa syntaxe est basée sur des balises.

Une balise dispose d'un nom, mis entre chevrons, et elle est répétée deux fois :

Code : XML

```
<nom></nom>
```

La première balise **<nom>** est la balise ouvrante, et la dernière balise **</nom>** est la balise fermante.

Il ne faut pas oublier le slash / pour signifier qu'il s'agit d'une balise fermante.

Il est possible d'imbriquer des balises dans d'autres :

Code : XML

```
<balise1>
    <balise2></balise2>
    <balise3>
        <balise4></balise4>
    </balise3>
</balise1>
```

Ici, la balise 4 est contenue dans la balise 3. Les balises 2 et 3 sont contenues dans la balise 1.

Attention à bien faire correspondre les noms des balises ouvrantes et fermantes !

Chaque balise ouvrante peut disposer d'attributs, ayant chacun un nom et une valeur entre guillemets doubles :

Code : XML

```
<nom attribut1="valeur" attribut2="valeur"></nom>
```

Si une balise ne contient rien, on peut utiliser sa forme raccourcie :

Code : XML

```
<balise/>
```

Reprenons l'exemple des balises imbriquées pour voir ce que cela donnerait :

Code : XML

```
<balise1>
    <balise2/>
    <balise3>
        <balise4/>
    </balise3>
</balise1>
```

Enfin, il est possible de mettre du texte brut à l'intérieur d'une balise :

Code : XML

```
<balise>Quel temps fera-t-il demain ?</balise>
```

On peut également mélanger des balises et du texte brut :

Code : XML

```
<balise1>Salut !<balise2>Comment va la famille ?</balise2></balise1>
```

L'HTML est donc du XML avec des balises spéciales que nous allons découvrir un peu plus loin.

Seule une toute petite partie de ces balises est supportée par la classe `TextField` ! Nous ne verrons que celles-ci, mais sachez qu'il en existe beaucoup d'autre. D'autre part, il peut y avoir quelques différences au niveau des attributs par rapport au "vrai" HTML.

Du HTML dans nos champs de texte

Il est possible d'utiliser une version très simplifiée de l'HTML dans nos champs de texte en utilisant l'attribut `htmlText` de la classe `TextField` à la place de l'attribut `text` que nous avons utilisé jusque là :

Code : Actionscript

```
// Activation du multi-ligne avant de sépicifier le texte HTML
monSuperTexte.multiline = true;

// Texte HTML
monSuperTexte.htmlText = 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit. In pharetra magna imperdiet elit pretium a
malesuada nisl pellentesque.';
```

Il faut mettre l'attribut `multiline` à **`true`** avant de spécifier le texte HTML, sinon les retours à la ligne seront ignorés, même si vous mettez `multiline` à **`true`** ensuite.

Balises principales

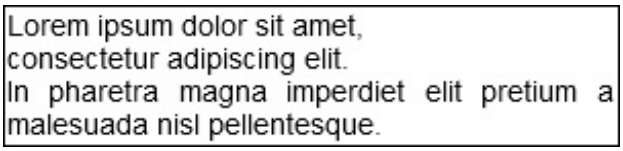
Retour à la ligne

Pour effectuer un retour à la ligne dans un texte HTML, nous utiliseront la balise `<br/>` comme ceci :

Code : Actionscript

```
monSuperTexte.htmlText = 'Lorem ipsum dolor sit
amet,<br/>consectetur adipiscing elit.<br/>In pharetra magna
imperdiet elit pretium a malesuada nisl pellentesque.';
```

Ce qui nous donne la figure suivante.



Deux retours à la ligne

Paragraphe

Un paragraphe est un bloc de texte pouvant être aligné, qui se termine par un retour à la ligne. La balise à utiliser est la balise `<p align="alignement">...</p>`.
L'attribut `align` peut prendre les valeurs suivantes :

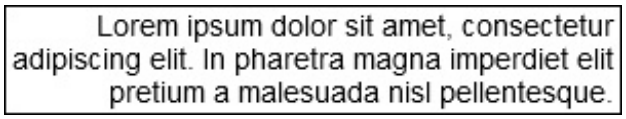
- `left` pour aligner le texte à gauche,
- `right` pour l'aligner à droite,
- `center` pour le centrer,
- `justify` pour le justifier.

Voici un exemple où notre texte est aligné à droite :

Code : Actionscript

```
monSuperTexte.htmlText = '<p align="right">Lorem ipsum dolor sit
amet, consectetur adipiscing elit. In pharetra magna imperdiet elit
pretium a malesuada nisl pellentesque.</p>';
```

Ce qui nous donne la figure suivante.



Paragraphe de texte aligné à droite

Voici un autre exemple avec plusieurs paragraphes :

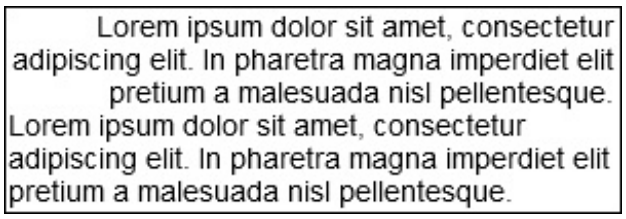
Code : Actionscript

```
monSuperTexte.htmlText = '<p align="right">Lorem ipsum dolor sit
amet, consectetur adipiscing elit. In pharetra magna imperdiet elit
pretium a malesuada nisl pellentesque.</p>' +
    '<p align="left">Lorem ipsum dolor sit amet, consectetur
adipiscing elit. In pharetra magna imperdiet elit pretium a
malesuada nisl pellentesque.</p>';
```

Vous remarquerez que j'ai utilisé la concaténation de chaînes de caractères avec l'opérateur + pour pouvoir écrire le

texte en plusieurs lignes.

Nous avons alors à l'écran la figure suivante.



Deux paragraphes alignés différemment

Lien

Un lien est une balise cliquable qui permet d'effectuer une action, le plus souvent permettant de changer de site web ou d'aller sur une autre page. Nous pouvons créer un lien à l'aide de la balise `<a href="adresse" target="cible">nom du lien</a>`.
L'attribut `href` contient l'adresse vers laquelle pointe le lien (par exemple, `http://www.siteduzero.com/`, l'adresse du Site du Zéro).
L'attribut `target` permet de spécifier la cible du lien, c'est à dire où l'adresse va s'ouvrir si l'on clique dessus. Il peut prendre les valeurs suivantes :

- `_self` : ouvre le lien dans la même page,
- `_blank` : ouvre le lien dans une nouvelle fenêtre, ou un nouvel onglet si le navigateur le permet.

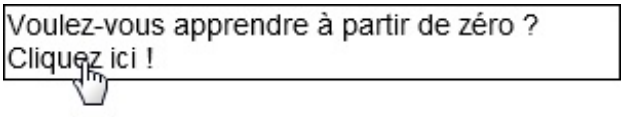
Bien sûr, ces valeurs n'ont d'intérêt que si votre animation Flash est incorporée dans un site web. Dans les autres cas (par exemple lorsque l'on teste le programme), cela ouvrira une nouvelle fenêtre (ou un nouvel onglet) dans votre navigateur, même si la valeur de l'attribut `target` est `_self`.

Voici un exemple de texte HTML disposant d'un lien pointant sur le Site du Zéro :

Code : Actionscript

```
monSuperTexte.htmlText = 'Voulez-vous apprendre à partir de zéro ?
<a href="http://www.siteduzero.com/" target="_blank">Cliquez ici
!</a>';
```

Ce qui nous donne la figure suivante.



Une partie du texte est cliquable

Pour tester l'attribut `target`, vous pouvez ouvrir votre fichier `swf` dans votre navigateur favori (à condition qu'il dispose du lecteur Flash). Vous verrez alors la page d'accueil du Site du Zéro se charger dans la même page ou s'ouvrir dans une nouvelle fenêtre ou un nouvel onglet

Image

Il est possible de charger une image locale dans un champ de texte grâce à la balise `<img src="adresse" width="largeur" height="hauteur" align="alignement" hspace="espacement horizontal" vspace="espacement vertical"/>`.

Essayons d'afficher Zozor dans notre champ de texte. Tout d'abord, téléchargez cette image et placez-là dans le dossier de votre fichier `swf` :

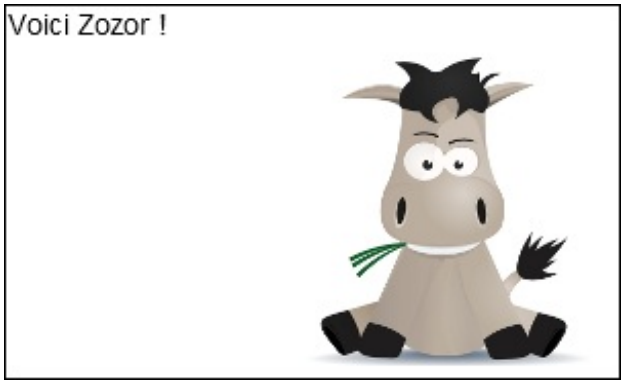
Zozor, la mascotte du Site du Zéro

Ensuite, renommez l'image en `zozor.png`.
Enfin, écrivons ce code pour charger l'image dans notre texte HTML :

Code : Actionscript

```
monSuperTexte.htmlText = 'Voici Zozor !<br/><img src="zozor.png"
width="150" height="150" align="right"/>';
```

Ce qui nous donne la figure suivante.



Zozor est dans notre champ de texte !

Il est impossible d'aligner une image au centre. Il s'agit d'une limitation du champ de texte, on ne peut rien y faire. 

Si on met un texte plus long à côté de l'image, il sera détouré. Pour illustrer ceci, prenons un texte beaucoup plus long à côté de Zozor, et alignons-le en justifié. :

Code : Actionscript

```
monSuperTexte.htmlText = '<img src="zozor.png" width="150"
height="150" align="right"/>' +
    '<p align="justify">Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Sed pharetra semper ullamcorper.' +
    'Sed at urna in elit mollis egestas eu id lectus. Integer
aliquam urna vitae massa varius sed cursus mauris aliquam.' +
    'Curabitur mauris massa, imperdiet a venenatis ac, vehicula eu
ipsum. Nulla turpis enim, tincidunt at imperdiet nec,' +
    'mollis vitae nunc. Morbi viverra iaculis neque et tincidunt.
Nullam ultrices mi mi, eu pellentesque leo.</p>' +
    '<p align="justify">Pellentesque sed felis nulla. Phasellus
laoreet sagittis ante at fringilla.' +
    'Duis iaculis, mi sed bibendum posuere, nisi velit cursus elit,
vitae tincidunt augue diam quis elit.' +
    'Sed et nisi risus. Vestibulum lobortis auctor viverra. Mauris
eleifend nisl ut orci ultricies eget egestas' +
    'ante lacinia. Pellentesque ut diam turpis.</p>';
```

Ce qui nous donne la figure suivante.

Cela ressemble à un article de journal n'est-ce pas ?

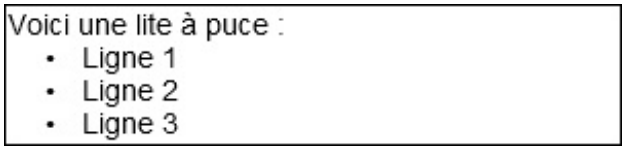
Liste à puce

Il est possible de faire des listes à puce grâce à la balise `<li>...</li>` répétée autant de fois qu'il y a de ligne dans la liste :

Code : Actionscript

```
monSuperTexte.htmlText = 'Voici une lite à puce :<li>Ligne
1</li><li>Ligne 2</li><li>Ligne 3</li>';
```

Ce qui nous donne la figure suivante.



Une liste à puce

Balises de mise en forme

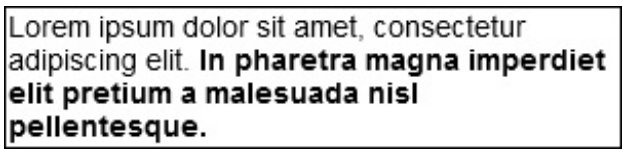
Gras

Pour mettre du texte en gras, rien de plus simple ! Il suffit de l'encadrer d'une balise `<b>...</b>` (b comme *bold*, signifiant *gras* en anglais) comme ceci :

Code : Actionscript

```
monSuperTexte.htmlText = 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit. <b>In pharetra magna imperdiet elit pretium a
malesuada nisl pellentesque.</b>';
```

Ce qui nous donne la figure suivante.



La deuxième phrase est en gras.

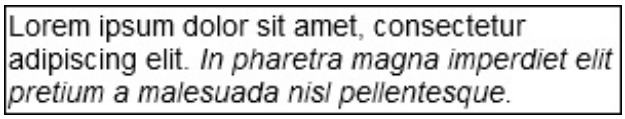
Italique

Pour mettre le texte en italique, c'est tout aussi simple : seule la balise change. Cette fois, il s'agit de `<i>...</i>` :

Code : Actionscript

```
monSuperTexte.htmlText = 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit. <i>In pharetra magna imperdiet elit pretium a
malesuada nisl pellentesque.</i>';
```

Ce qui nous donne la figure suivante.



La deuxième phrase est en italique.

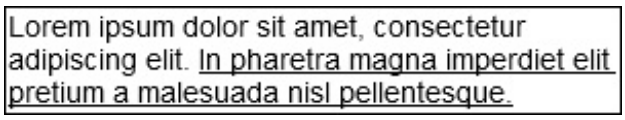
Souligné

Il est possible de souligner du texte grâce à la balise `<u>...</u>` comme ceci :

Code : Actionscript

```
monSuperTexte.htmlText = 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit. <u>In pharetra magna imperdiet elit pretium a
malesuada nisl pellentesque.</u>';
```

Ce qui nous donne la figure suivante.



La deuxième phrase est soulignée.

Police, couleur et taille

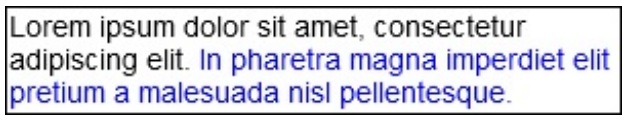
Pour modifier la police de caractère, la couleur ou la taille des caractères du texte, la balise `<font face="police" color="couleur" size="taille">...</font>` est tout indiquée.
L'attribut `face` peut recevoir comme valeur le nom d'une police de caractère, comme par exemple `face="Arial"`.
L'attribut `color` permet de définir une couleur en hexadécimal, précédée d'un dièse. Exemple de couleur orange : `color="#ff8800"`.
Enfin, l'attribut `size` modifie la taille des caractères du texte. On peut définir une taille absolue en pixels, ou une taille relative en ajoutant un + ou un - devant. Exemple de taille absolue : `size="42"`. Exemple de taille relative : `size="-5"`, ce qui signifie la taille actuelle du texte moins cinq pixels.

Par exemple, nous pouvons colorier une partie de notre texte en bleu :

Code : Actionscript

```
monSuperTexte.htmlText = 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit. <font color="#0000ff">In pharetra magna imperdiet
elit pretium a malesuada nisl pellentesque.</font>';
```

Ce qui nous donne la figure suivante.



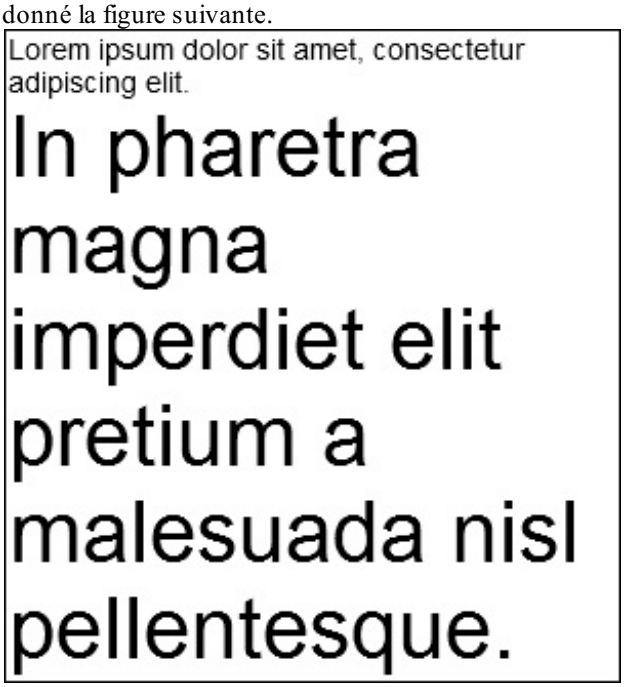
La deuxième phrase en bleu !

Nous aurions pu, au lieu de cela, augmenter la taille de notre texte :

Code : Actionscript

```
monSuperTexte.htmlText = 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit.<br/><font size="42">In pharetra magna imperdiet
elit pretium a malesuada nisl pellentesque.</font>';
```

Ce qui aurait donné la figure suivante.



La deuxième phrase en taille 42 cette fois-ci.

Les polices de caractères embarquées

Embarquer des polices

Pourquoi ?

Jusqu'à présent, les polices que nous utilisons sont des polices fournies par le système d'exploitation sur lequel notre programme est lancé. Si nous vouons appliquer une police particulière à notre texte, mais qu'elle n'ai pas présente sur le système, la police par défaut sera utilisée. De plus, il nous est impossible de faire tourner notre texte ou de changer son opacité, ce qui est un peu dommage !

Cependant, il existe une solution à ces problèmes : les polices de caractères embarquées !
Il s'agit d'inclure un fichier contenant une police directement dans notre programme : ainsi, le lecteur Flash peut la rendre directement à l'écran, même si le système de dispose pas de cette police en particulier ! De plus, cela permet de lisser les caractères du texte afin de les rendre plus lisibles et plus attractifs.

Bien évidemment, embarquer des fichiers directement dans le programme augmente sa taille. Cela peut éventuellement être gênant, mais embarquer les polices de caractère présente tant d'avantages que l'on passera souvent outre cet unique inconvénient.

Préparation

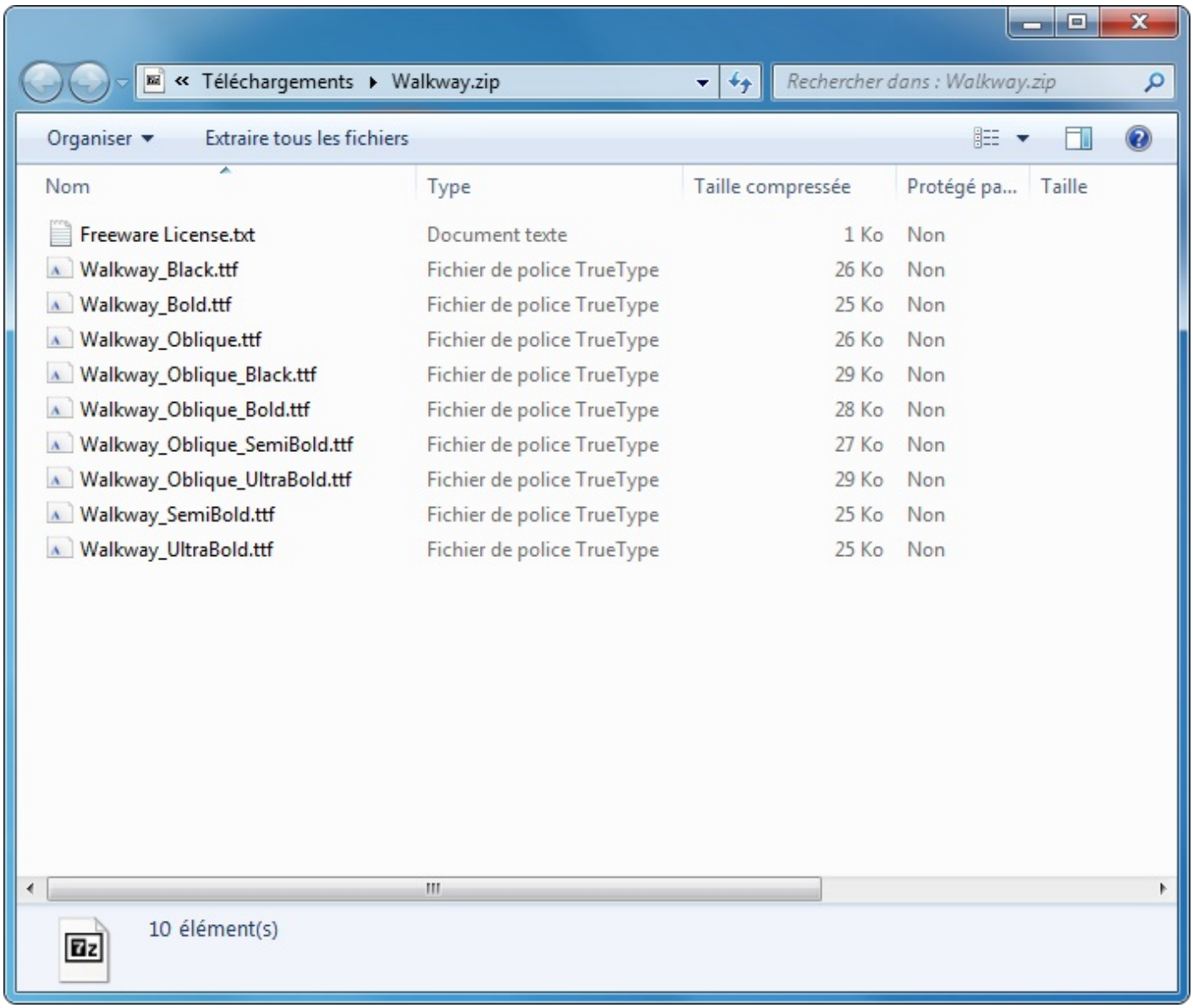
Tout d'abord, il faut se munir du fichier de la police de caractère à embarquer dans notre programme. Seules les polices de format *TrueType* sont reconnues, mais c'est le format le plus répandu, donc cela pose rarement problème.

Par exemple, vous pouvez prendre une police installée sur votre système.
Sur Windows, les polices sont situées dans le dossier `C:\Windows\Fonts`.
Sur Mac, il peut y en avoir à plusieurs endroits : `~/Library/Fonts/`, `/Library/Fonts/`, `/Network/Library/Fonts/`, `/System/Library/Fonts/` ou `/System Folder/Fonts/`.
Enfin, sur Linux, on peut les trouver dans ces dossiers : `/usr/share/fonts`, `/usr/local/share/fonts` ou `~/.fonts`.

Ou bien vous pouvez vous rendre sur un site proposant gratuitement des polices de caractères, comme FontSquirrel.
Prenons par exemple la police *Walkway* (voir figure suivante). Téléchargez-la sur le site en cliquant sur le bouton *DOWNLOAD TTF*.

Page de téléchargement de la police Walkway

Vous obtenez une archive zip, contenant différents fichiers de police (voir figure suivante).



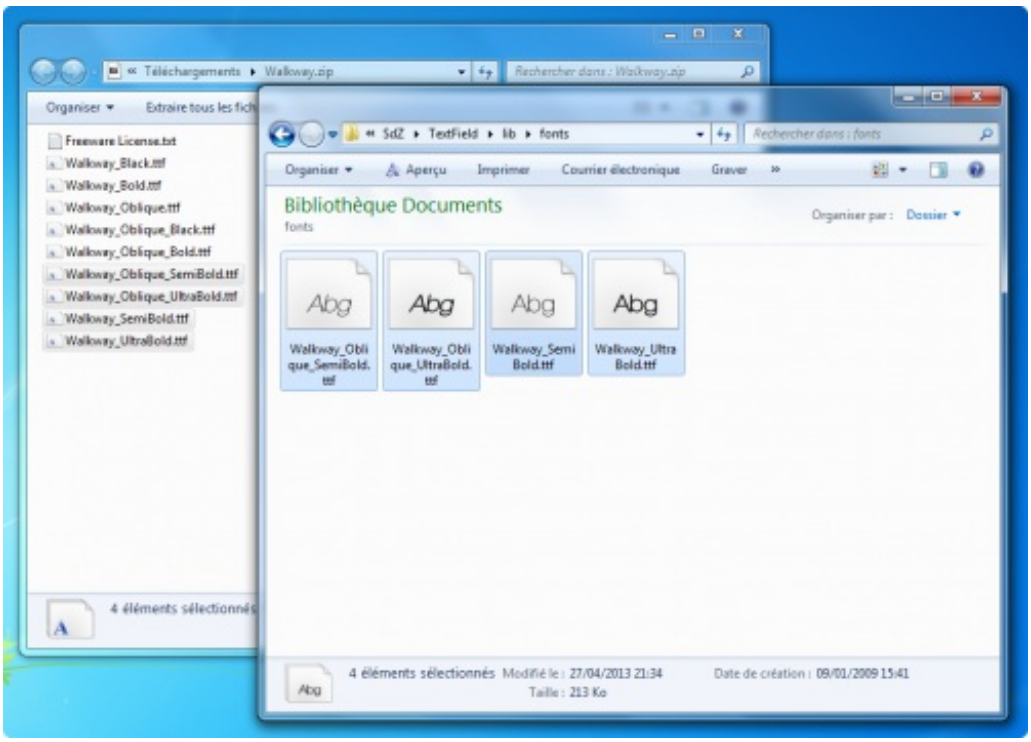
Contenu de l'archive téléchargée

Nous aurons besoins de quatre de ces fichiers, un par style de notre police, comme le montre le tableau suivant.

Style	Nom du fichier
Normal	Walkway_SemiBold.ttf
Gras	Walkway_UltraBold.ttf
Italique	Walkway_Oblique_SemiBold.ttf
Gras Italique	Walkway_Oblique_UltraBold.ttf

Vous pouvez incorporer une police de caractères dont il manque certains styles, mais vous ne pourrez pas utiliser les styles manquant. Par exemple, si vous n'avez pas le fichier pour l'italique, vous ne pourrez pas mettre de texte en italique avec cette police.

Dans le répertoire du projet, créez un dossier `lib` s'il n'est pas déjà présent, puis créez un dossier à l'intérieur nommé `fonts`. Enfin, extrayez les quatre fichiers dont nous avons besoin dans ce dossier (voir figure suivante).



Les fichiers de police prêts à l'emploi

Implémentation

Une fois que nos fichiers sont prêts, il faut les incorporer à notre programme, puis les enregistrer sous le nom `"walkway"` pour pouvoir utiliser la police de caractère embarquée partout dans notre projet.
Créez une nouvelle classe `EmbedFonts` (dans un fichier nommé `EmbedFonts.as`) à côté de la classe `Main` et collez-y le code complet suivant :

Code : Actionscript

```
package
{
    import flash.text.Font;

    /**
* Gestionnaire des polices de caractères embarquées.
* Pour ajouter une police, placer ces fichiers dans le dossier lib/fonts (par ex
classe EmbedFonts ainsi :
* [Embed(source='../lib/fonts/verdana.ttf',fontName='Fverdana',fontWeight="norma
* private static var F_VERDANA:Class;
* puis enregistrez-la dans la méthode init() ainsi :
* Font.registerFont(F_VERDANA);
* @author Guillaume CHAU
*/
    public class EmbedFonts
    {

        // Incorporation des polices de caractères

        ///// Walkway
        // Normal
        [Embed(source='../lib/fonts/Walkway_SemiBold.ttf',fontName='walkway',for
        private static var F_WALKWAY:Class;
        // Gras
        [Embed(source='../lib/fonts/Walkway_UltraBold.ttf',fontName='walkway',fo
        private static var F_WALKWAY_BOLD:Class;
        // Italique
        [Embed(source='../lib/fonts/Walkway_Oblique_SemiBold.ttf',fontName='walk
        private static var F_WALKWAY_ITALIC:Class;
        // Gras Italique
```

```
    [Embed(source='../lib/fonts/Walkway_Oblique_UltraBold.ttf',fontName='walkway',fo
            private static var F_WALKWAY_BOLD_ITALIC:Class;

            /**
    * Initialisation des polices de caractères. A appeler une fois au lancement de l
    compte.
    */
            public static function init():void
            {
                // Enregistrement des polices de caractères

                Font.registerFont(F_WALKWAY);
                Font.registerFont(F_WALKWAY_BOLD);
                Font.registerFont(F_WALKWAY_BOLD_ITALIC);
                Font.registerFont(F_WALKWAY_ITALIC);
            }

        }

    }
```

Nous commençons par importer chaque fichier au début de la classe grâce une instruction spéciale, destinée au compilateur :

Code : Actionscript

```
[Embed(source='../lib/fonts/Walkway_SemiBold.ttf',fontName='walkway',fontWeight='
```

J'ai l'impression d'avoir déjà vu quelque chose de similaire...

Très juste ! Nous avons déjà écrit une instruction spéciale dans notre classe `Main` pour régler la taille et la couleur de fond de notre fenêtre :

Code : Actionscript

```
[SWF(width="640",height="480",backgroundColor="#ffffff")]
```

Ici, le mot-clé `SWF` signifie que nous allons modifier des propriétés de notre programme.

Cette-fois, le mot-clé `Embed` signifie que nous allons inclure un fichier dans notre programme. Les fichiers inclus sont directement contenus dans le programme : il sont donc utilisables à tout moment, étant chargés en même temps.

Code : Actionscript

```
source='../lib/fonts/Walkway_SemiBold.ttf'
```

L'attribut `source` indique le chemin du fichier à inclure, ici `../lib/fonts/Walkway_SemiBold.ttf`.

Code : Actionscript

```
fontName='walkway'
```

L'attribut `fontName` spécifie le nom de la police de caractère, nom qui nous servira à l'utiliser ensuite.

Code : Actionscript

```
fontWeight="normal"
```

Les attributs `fontStyle` (style) et `fontWeight` (épaisseur) permettent de préciser de quel style il s'agit. En combinant les deux, il est possible de créer un style *gras italique* :

Code : Actionscript

```
fontStyle='italic',fontWeight='bold'
```

Enfin, le dernier attribut `embedAsCFF` est obligatoire et doit être mis à `false`. Cela permet d'éviter tout problème lors de l'utilisation de polices embarquées, à cause d'un changement entre *Flex SDK 3* et *Flex SDK 4*.

Sur la ligne suivante, nous déclarons un attribut privé de type `Class` qui contiendra le fichier de police embarqué :

Code : Actionscript

```
private static var F_WALKWAY:Class;
```

Ce qui nous donne pour nos quatre fichiers :

Code : Actionscript

```
///// Walkway
// Normal
[Embed(source='../lib/fonts/Walkway_SemiBold.ttf',fontName='walkway',fontWeight=
private static var F_WALKWAY:Class;
// Gras
[Embed(source='../lib/fonts/Walkway_UltraBold.ttf',fontName='walkway',fontWeight
private static var F_WALKWAY_BOLD:Class;
// Italique
[Embed(source='../lib/fonts/Walkway_Oblique_SemiBold.ttf',fontName='walkway',for
private static var F_WALKWAY_ITALIC:Class;
// Gras Italique
[Embed(source='../lib/fonts/Walkway_Oblique_UltraBold.ttf',fontName='walkway',fo
private static var F_WALKWAY_BOLD_ITALIC:Class;
```

Une fois que tous les fichiers sont incorporés, nous déclarons une méthode publique statique `init():void` qui nous permet d'enregistrer la police de caractère, qui sera appelée au tout début de notre programme :

Code : Actionscript

```
/**
* Initialisation des polices de caractères. A appeler une fois au
lancement de l'application, afin que les polices soient prises en
compte.
*/
public static function init():void
{
 // Enregistrement des polices de caractères
}
```

Dans cette méthode, nous enregistrons les classes contenant les fichiers des polices de caractères grâce à la méthode statique

`registerFont` de la classe `Font` :

Code : Actionscript

```
Font.registerFont(F_WALKWAY);
```

Ce qui donne :

Code : Actionscript

```
/**
* Initialisation des polices de caractères. A appeler une fois au
lancement de l'application, afin que les polices soient prises en
compte.
*/
public static function init():void
{
 // Enregistrement des polices de caractères

 Font.registerFont(F_WALKWAY);
 Font.registerFont(F_WALKWAY_BOLD);
 Font.registerFont(F_WALKWAY_BOLD_ITALIC);
 Font.registerFont(F_WALKWAY_ITALIC);
}
```

Enfin, dans notre classe `Main`, nous appelons la méthode statique `init` de notre classe `EmbedFonts` :

Code : Actionscript

```
package
{
    import flash.display.Sprite;
    import flash.display.StageAlign;
    import flash.display.StageScaleMode;
    import flash.events.Event;

    /**
* Notre application embarque des polices de caractères embarquées !
:)
*/
    [SWF(width="640",height="480",backgroundColor="#ffffff")]

    public class Main extends Sprite
    {

        public function Main():void
        {
            if (stage)
                init();
            else
                addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void
        {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // entry point

            stage.scaleMode = StageScaleMode.NO_SCALE;
            stage.align = StageAlign.TOP_LEFT;

            // Polices embarquées
            EmbedFonts.init();
        }
```

```
        }

    }
```

Vous remarquerez qu'il n'est pas nécessaire d'importer la classe `EmbedFonts` car elle est dans le *package* principal (elle est placée dans le dossier racine de la source, ici `src/`).

Utilisation

Et voilà ! Notre police de caractère embarquée est prête à l'emploi ! 
Pour que notre champ de texte utilise les polices de caractères embarquées au lieu des polices du système d'exploitation, il faut mettre l'attribut `embedFonts` de la classe `TextField` à **`true`**, sans oublier de changer le nom de la police utilisée :

Code : Actionscript

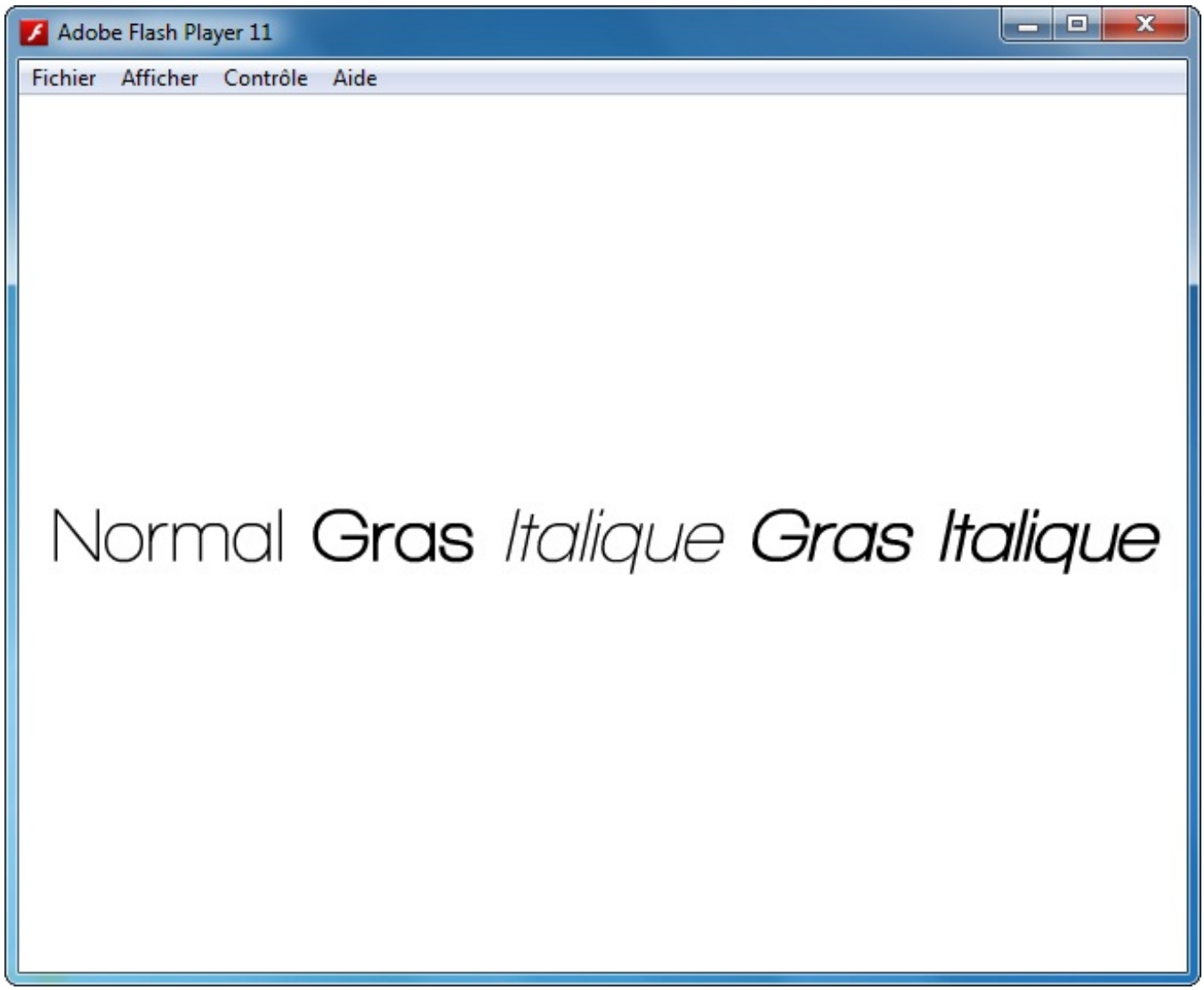
```
// Police utilisée : walkway
var format:TextFormat = new TextFormat("walkway", 42, 0x000000);
monSuperTexte.defaultTextFormat = format;

// On utilise les polices embarquées
monSuperTexte.embedFonts = true;
```

Voici le code complet pour afficher un champ de texte avec une police de caractères embarquée :

Code : Actionscript

```
// Enregistrement des polices embarquées
EmbedFonts.init();

// Création de l'instance du champ de texte
var monSuperTexte:TextField = new TextField();

// Police utilisée : walkway
var format:TextFormat = new TextFormat("walkway", 42, 0x000000);
monSuperTexte.defaultTextFormat = format;

// On utilise les polices embarquées
monSuperTexte.embedFonts = true;

// Taille automatique
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;

// Texte HTML
monSuperTexte.htmlText = "Normal <b>Gras</b> <i>Italique</i>
<b><i>Gras Italique</i></b>";

// Ajout à la scène
addChild(monSuperTexte);

// Alignement au centre
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;
```

Ce qui nous donne à l'écran la figure suivante.



Notre police embarquée en action !

Désormais, notre texte s'affichera avec cette police même si l'utilisateur ne l'a pas installée ; de plus, il est possible de rendre le texte transparent et de lui appliquer une rotation !

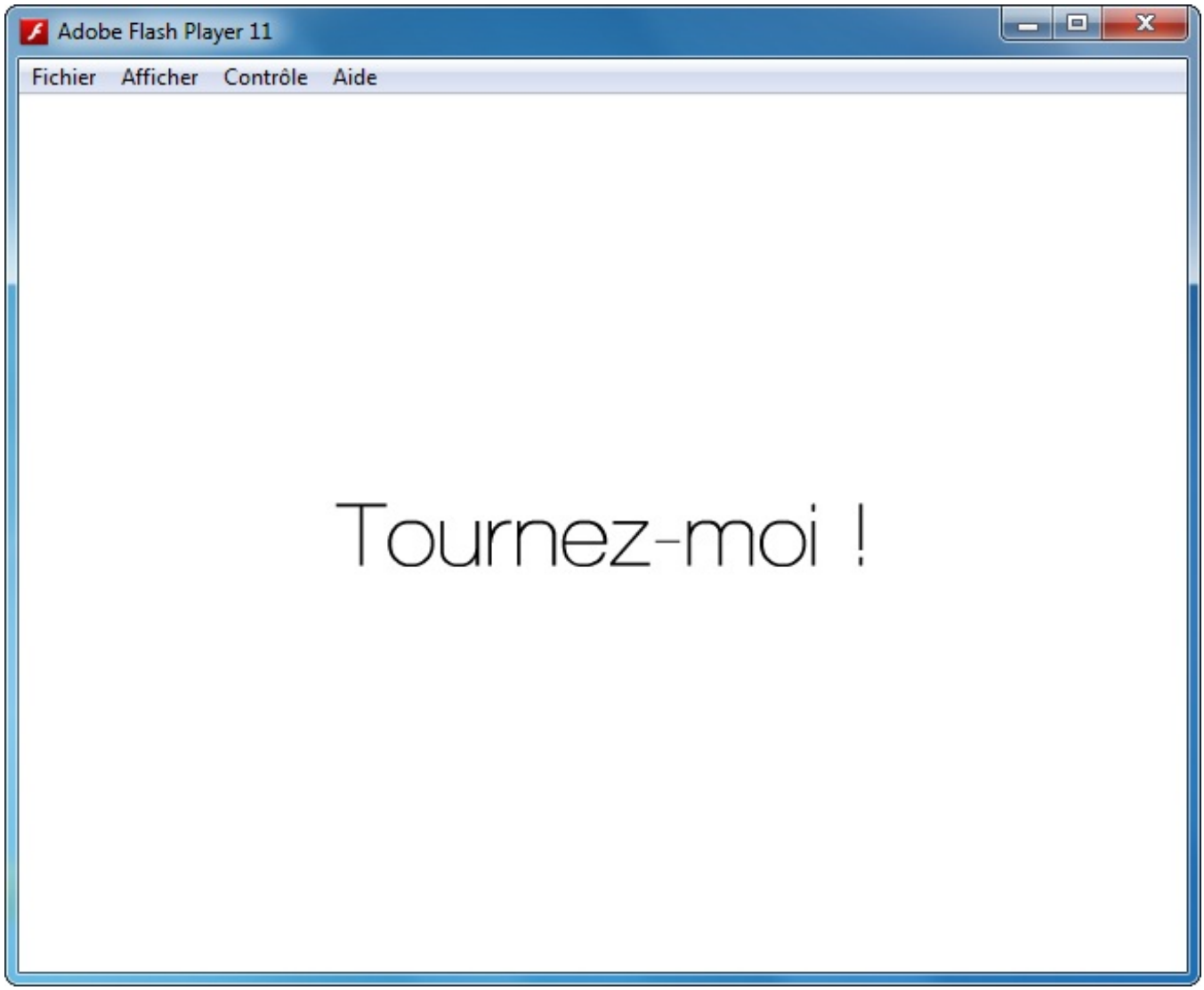
Rotation sur soi-même

Et si nous décidions de tourner notre champ de texte sur lui-même ? Disons 50° dans le sens anti-horaire ?
C'est désormais possible, tant donné que nous utilisons notre propre police embarquée.
Commençons par écrire le code pour créer un champ de texte utilisant une police de caractères embarquée :

Code : Actionscript

```
// Polices embarquées
EmbedFonts.init();

// Création du champ de texte
var monSuperTexte:TextField = new TextField();
var format:TextFormat = new TextFormat("walkway", 50, 0x000000);
monSuperTexte.defaultTextFormat = format;
monSuperTexte.embedFonts = true;
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;
monSuperTexte.text = "Tournez-moi !";
addChild(monSuperTexte);

// Alignement au centre
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;
```

Ce qui nous donne la figure suivante.



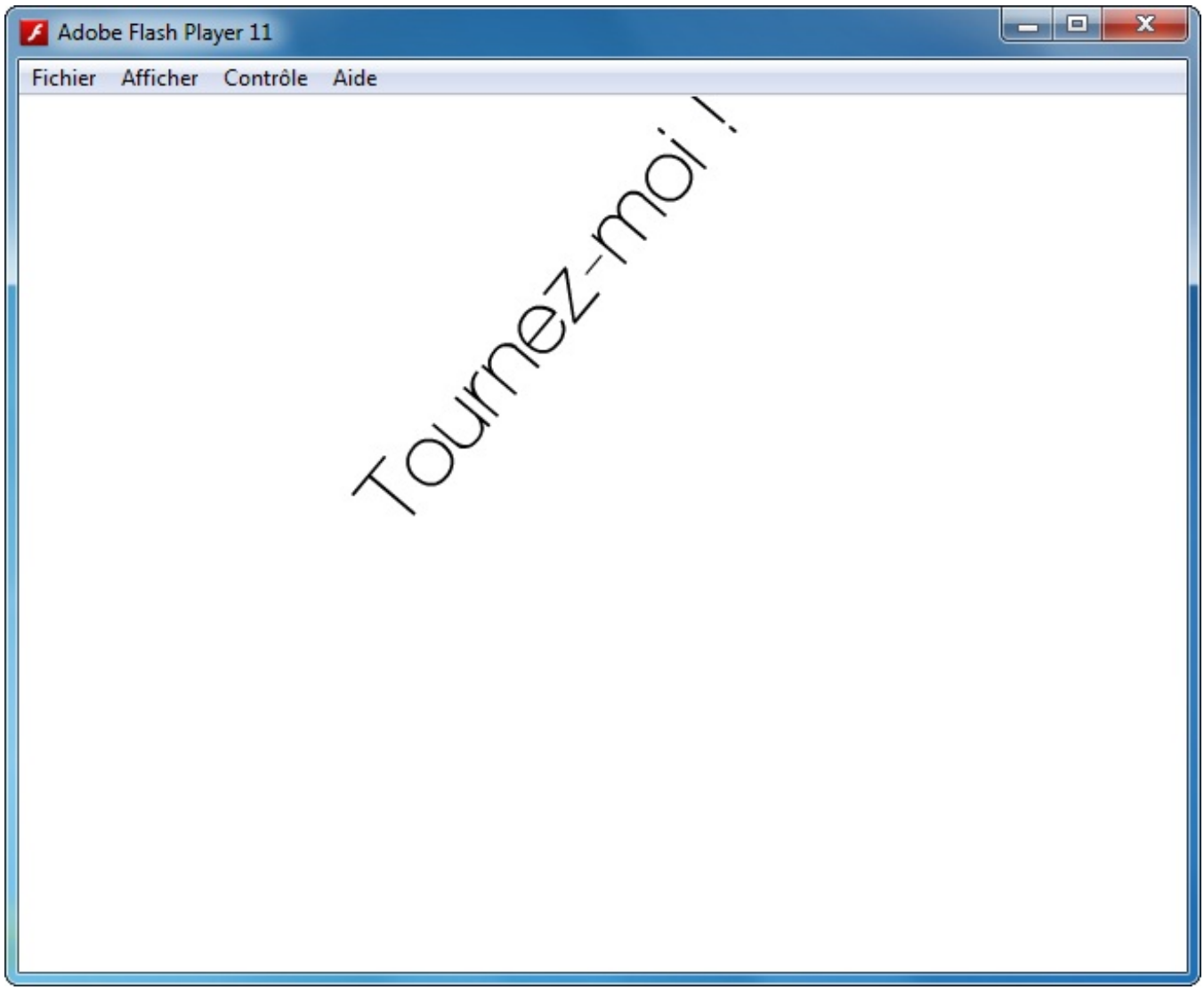
Notre champ de texte à tourner

Essayons d'effectuer une rotation à notre champ de texte :

Code : Actionscript

```
// Rotation de 50° dans le sens anti-horaire
monSuperTexte.rotation = - 50;
```

Ce qui nous donne la figure suivante.



Rotation par rapport à l'origine de l'objet

Ce n'est pas exactement ce que nous voulons accomplir ! En effet, l'objet tourne toujours par rapport à son origine, en haut à gauche : cela explique que le champ de texte soit tourné par rapport à la lettre "T". Malheureusement, nous ne pouvons pas modifier directement l'origine d'un objet d'affichage.

Il est néanmoins possible de changer artificiellement l'origine de l'objet en question, comme nous l'avons vu dans le chapitre précédent, en l'ajoutant dans un conteneur intermédiaire de classe `Sprite` que j'appellerai "la boîte" :

Code : Actionscript

```
// Conteneur intermédiaire
var boite:Sprite = new Sprite();
addChild(boite);

// Création du champ de texte
var monSuperTexte:TextField = new TextField();
var format:TextFormat = new TextFormat("walkway", 50, 0x000000);
monSuperTexte.defaultTextFormat = format;
monSuperTexte.embedFonts = true;
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;
monSuperTexte.text = "Tournez-moi !";

// On ajoute le champ de texte dans le conteneur
boite.addChild(monSuperTexte);
```

Une fois que le champ de texte est dans la boîte, on déplace le champ de texte de la moitié de sa taille, à la fois vers le haut et vers

la gauche, ainsi l'origine de la boite se trouvera au centre du champ de texte (voir figure suivante).

Origine de la boite centrée sur le champ de texte

Ainsi, l'origine de la boite se retrouve au centre du champ de texte. Si on fait abstraction de la boîte, l'origine de notre objet est le centre du champ de texte, donc il tournera par rapport à son centre !
Voici le code de cette opération :

Code : Actionscript

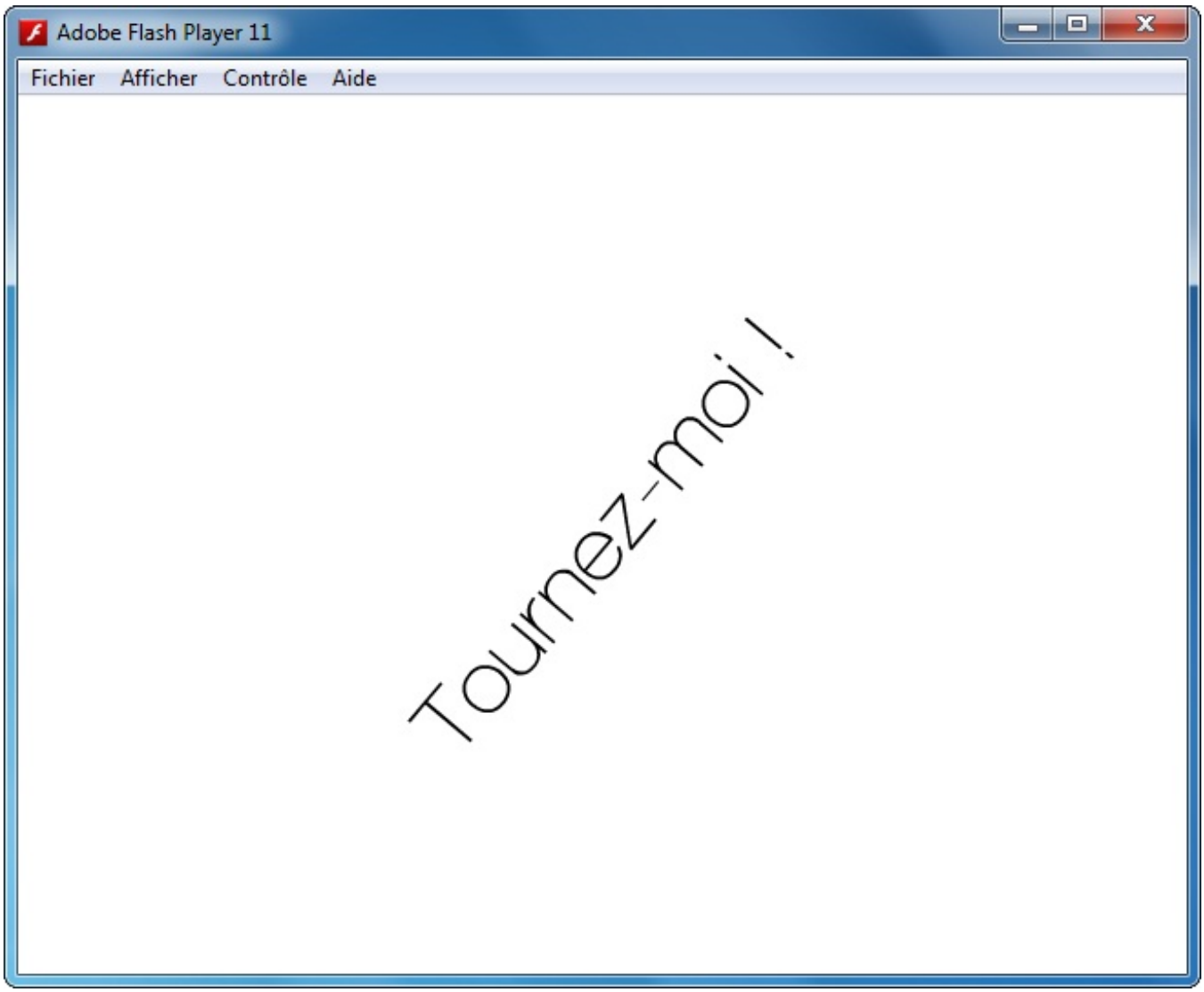
```
// On modifie artificiellement l'origine en la plaçant au centre du
champ de texte :
monSuperTexte.x = -monSuperTexte.width / 2;
monSuperTexte.y = -monSuperTexte.height / 2;
```

Enfin, on applique toutes les transformations à la boîte à la place de les appliquer au champ de texte, comme si la boîte **était** le champ de texte lui-même :

Code : Actionscript

```
// Alignement au centre
// On aligne le conteneur au lieu du champ de texte
boite.x = stage.stageWidth / 2;
boite.y = stage.stageHeight / 2;

// Rotation de 50° dans le sens anti-horaire
// On tourne le conteneur au lieu du champ de texte
boite.rotation = - 50;
```

Ce qui nous donne ce que nous voulons (voir figure suivante).



Le texte tourne sur lui-même !

Enfin ! Notre champ de texte a enfin consenti à tourner sur lui-même par rapport à son centre ! 

Vous serez souvent amené à utiliser cette méthode, par exemple si vous voulez animer une image en la faisant tourner sur elle-même, tout en la déplaçant...

Vous auriez peut-être pensé à recentrer le champ de texte après chaque rotation, mais croyez-moi, il est beaucoup plus simple de passer par un conteneur intermédiaire. 

En résumé

- La classe `TextField` permet d'afficher du texte à l'écran. Il est possible de manipuler la sélection du texte.
- On peut centrer un objet grâce aux propriétés de la scène principale de classe `Stage`.
- La classe `TextFormat` permet de mettre en forme tout ou des portions du texte, c'est-à-dire change la police, la taille, la couleur...
- Il est possible d'afficher du texte structuré en HTML simplifié avec l'attribut `htmlText`.
- Il est très souvent intéressant d'embarquer les polices de caractères dans l'application, sans oublier de mettre l'attribut `embedFonts` à **`true`**.
- Utiliser un conteneur intermédiaire permet d'effectuer la rotation d'un objet autour de son centre (ou d'un autre point que son origine initiale).

Dessiner avec l'Actionscript

Dans ce chapitre, nous allons apprendre à dessiner directement depuis l'Actionscript ! Je suis certain que beaucoup parmi vous attendaient ce moment depuis fort longtemps, c'est pourquoi nous essaierons de présenter un maximum de choses tout en prenant le temps de bien expliquer. Nous présenterons en premier la classe `Graphics`, qui est à l'origine de tout tracé en Actionscript. Puis nous présenterons alors les différentes méthodes de dessin que possède cette classe.

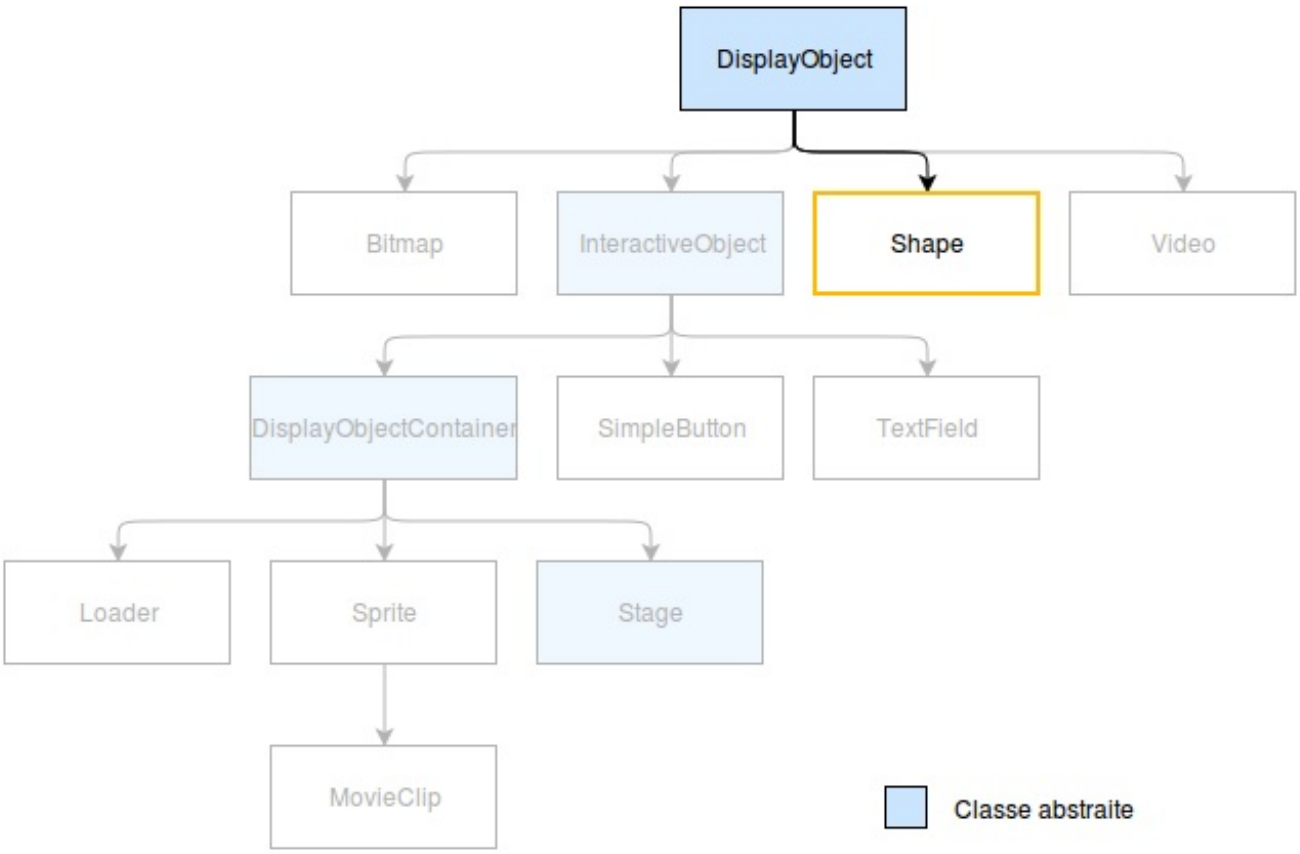
Enfin pour finir, je vous propose un petit exercice où nous réaliserons ensemble une illustration d'un mouton. Ce sera alors l'occasion de revenir sur ces méthodes de dessin et mieux comprendre toute la philosophie qui se cache derrière le dessin en Flash !

L'objet Graphics

Introduction

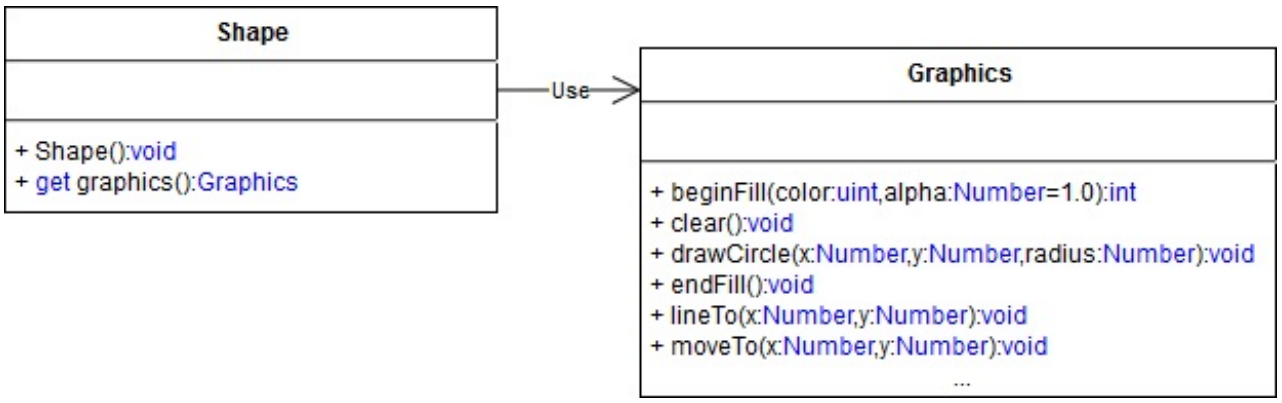
Une propriété de type `Graphics`

Nous avons vu précédemment qu'il existait différents types d'objets d'affichage. Parmi eux, certains possèdent déjà tout le nécessaire pour dessiner. Il s'agit des classes `Shape`, `Sprite` et `MovieClip` (voir figure suivante) !



La classe Shape dans l'arbre des classes d'affichage

Ce qui les différencie des autres classes c'est que celles-ci ont un attribut très particulier : une instance de la classe `Graphics`. En réalité, c'est cette dernière classe qui dispose de l'ensemble des méthodes permettant de tracer différentes formes. Ainsi en déclarant des objets de type `Shape`, `Sprite` ou `MovieClip` vous pourrez dessiner à l'intérieur de ceux-ci. En particulier, la classe `Shape` est spécialement faite pour cela puisque celle-ci ne contient que cet attribut et l'accesseur correspondant comme vous pouvez le voir à la figure suivante?



La classe Shape exclusivement utilisé pour sa propriété graphics

Cette classe est allégée, ce qui implique de meilleures performances, mais une instance de la classe `Shape` ne peut pas contenir d'enfants et elle n'est pas interactive (il est impossible de savoir si l'utilisateur clique dessus).

Pourquoi ne pas utiliser directement l'objet `Graphics` ?

Il se peut que vous vous soyez déjà demandé pourquoi ne pas créer directement une occurrence de la classe `Graphics` et utiliser ainsi ses différentes méthodes de dessin. En réalité, l'Actionscript favorise l'utilisation des sous-classes de `DisplayObject`, dont n'hérite pas la classe `Graphics`. C'est pourquoi celle-ci est non instanciable : vous ne pourrez donc pas utiliser la syntaxe « `new Graphics()` » et vous serez donc obligés de passer par l'utilisation de l'une des trois classes décrites plus haut. De plus, elle est de type **`final`**, ce qui signifie que l'on ne peut pas créer de sous-classes héritant de la classe `Graphics`.

Je vous invite donc à créer une instance de la classe `Shape` que nous utiliserons dans la suite :

Code : Actionscript

```
var monDessin:Shape = new Shape();
```

Dessin vectoriel

La classe `Graphics` est utilisée pour créer des **dessins vectoriels** !
Étant donné que ce terme est peut-être nouveau pour vous, nous allons ici présenter ce qui différencie ce type d'images par rapport aux images « bitmap ». Vous ne vous êtes jamais demandés comment les images étaient stockées à l'intérieur de votre ordinateur ?
En fait le stockage des images peut être réalisé de deux manières différentes. Lorsqu'on parle d'images « bitmap », celles-ci sont enregistrées à partir des couleurs de chacun des pixels. En revanche dans les images vectorielles, ce sont les formes en elles-mêmes qui sont sauvegardées. Cela fait donc une énorme différence lors de la lecture du fichier et de l'affichage à l'écran.

Pour mieux comprendre, nous allons prendre l'exemple de la figure suivante.


Image de base

Cette image est quelconque et pourrait très bien être enregistrée en tant qu'image « bitmap » ou vectorielle. L'inconvénient du format « bitmap » est justement qu'il retient des pixels. Ainsi si vous respectez plus les dimensions des pixels, l'image va être déformée. Je m'explique ; imaginez que vous ayez besoin, pour une raison quelconque, d'agrandir votre image. Les pixels vont donc être redimensionnés et l'image n'aura plus l'aspect lisse qu'elle avait au départ. Au contraire, une image vectorielle est définie à partir de formes. Ainsi lorsque celle-ci va être redimensionnée, l'affichage sera adapté pour correspondre à nouveau à la forme désirée.

Voyez plutôt la différence à la figure suivante entre l'image « bitmap » à gauche et l'image vectorielle à droite, toutes les deux redimensionnées à 300%.

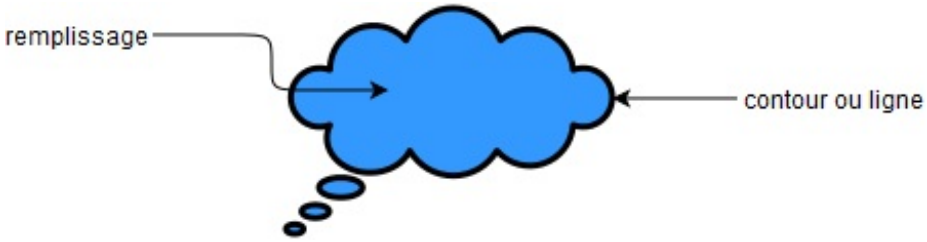
Redimensionnement de l'image e base

Dans ce chapitre, nous allons donc parler exclusivement de dessin vectoriel, qui l'un des atouts de cette technologie Flash !

Des contours et des remplissages

Le principe

Si vous avez déjà utilisé des classes de dessin dans d'autres langages de programmation, vous allez voir qu'ici le principe est assez différent. Avant d'entrer plus dans les détails, je vous invite à découvrir les différents termes que nous serons amenés à utiliser dans la suite de ce chapitre. Pour cela je vous propose tout d'abord un petit dessin d'exemple à la figure suivante.



Composition d'un élément graphique

Dans cet exemple nous pouvons voir une bulle de pensée comme on en trouve dans les bandes dessinées. Cette bulle est en fait composée de 4 objets : un « nuage » et trois ovales. Ces objets correspondent en réalité à des tracés différents, qu'il faudrait réaliser l'un après l'autre.
Chaque tracé est donc défini de la façon suivante :

- le **remplissage** : surface représentée en bleu sur le dessin précédent
- le **contour** : ligne qui délimite le remplissage
- la **ligne** : trait visible qui suit le contour et pouvant avoir différentes apparences.

Ainsi pour réaliser un tracé en Flash, il nous faut définir le contour de celui-ci en indiquant quel style adopter pour le remplissage et la ligne.

Définir le style des lignes

Tout d'abord, veuillez noter que nous parlons ici de *lignes* et non de *contours*, il s'agit donc uniquement du côté esthétique des traits qui effectivement suivent le contour.
Pour définir le style des lignes, nous utiliserons la méthode `lineStyle()` de la classe `Graphics`. Cette méthode possède beaucoup de paramètres, mais tous sont facultatifs. C'est pourquoi je ne vous parlerai ici que des trois premiers qui sont les plus intéressants. Le premier sert à définir la largeur de la ligne, le deuxième sa couleur, et enfin de dernier permet de préciser l'opacité de celle-ci.
Voici un exemple dans lequel nous définissons des lignes de largeur 2, de couleur noire et transparentes à 50% :

Code : Actionscript

```
monDessin.graphics.lineStyle(2, 0x000000, 0.5);
```

Comme nous l'avons dit, la méthode `lineStyle()` ne possède que des paramètres facultatifs. Ainsi si vous ne souhaitez pas de lignes sur votre tracé mais uniquement un remplissage, il vous suffit d'utiliser cette méthode sans paramètres, comme ceci :

Code : Actionscript

```
monDessin.graphics.lineStyle();
```

Définir un remplissage

Contrairement aux lignes, les remplissages nécessitent l'utilisation de deux méthodes : `beginFill()` et `endFill()`.
La première permet de définir les différentes caractéristiques du remplissage, à savoir sa couleur et son opacité. En revanche ici la couleur est un paramètre obligatoire et ne peut donc pas être omis. Par exemple, voici comment définir une couleur blanche au remplissage :

Code : Actionscript

```
monDessin.graphics.beginFill(0xFFFFFF);
```

L'utilisation de la seconde méthode est justifiée par la définition même d'un remplissage. En effet, pour remplir l'intérieur d'un contour, celui-ci doit être *fermé* !
La méthode `endFill()` sert donc à refermer un contour, pour pouvoir lui appliquer un remplissage. Ainsi si lors du tracé de votre contour, le point final n'est identique au premier, une ligne droite alors tracée entre ces deux points pour fermer le contour.
Cette méthode `endFill()` ne prend aucun paramètre et s'utilise donc simplement de la manière suivante :

Code : Actionscript

```
monDessin.graphics.endFill();
```

Notez que la méthode `beginFill()` doit être appelée avant `endFill()`, et que la définition du contour s'effectue entre ces instructions. Également, tout tracé effectué à l'extérieur de ces deux instructions sera composé uniquement de lignes, et donc sans remplissage.

Dessinez, c'est gagné !

Ça y est, vous allez enfin pouvoir dessiner en Actionscript ! 
Nous allons maintenant voir quels sont les différentes méthodes de la classe `Graphics`, et apprendre à nous en servir. Ensuite nous réaliserons un petit exercice à la fin de ce chapitre, qui devrait vous aider à mieux utiliser cette classe.

Les lignes et les courbes

Une histoire de curseur

À l'origine, l'ensemble des tracés en Flash étaient conçus à l'intérieur du logiciel Flash Professionnal. Ceux-ci étaient donc réalisés graphiquement à l'aide de votre souris, aussi connue sous le nom de *curseur*. Depuis l'arrivée de l'Actionscript 3, il n'est plus nécessaire de posséder le logiciel pour pouvoir effectuer des tracés. En effet, il est maintenant possible de faire exactement la même chose grâce au code. Cependant les instructions s'exécutent en conservant la logique utilisée pour dessiner dans le logiciel.

Ainsi lorsque vous dessinerez en Actionscript, vous devrez imaginez les différents gestes que vous réaliseriez avec la souris de votre ordinateur. Par exemple si vous souhaitiez tracer une ligne, vous déplaceriez d'abord votre curseur à l'endroit où vous voudriez démarrer celle-ci, puis vous la traceriez. Une fois celle-ci dessinée, votre curseur se trouverait alors à l'autre extrémité de la ligne. Vous pourriez alors continuer votre tracé à partir du même point, ou bien déplacer avant votre curseur vers une autre partie de l'écran.

Revenons maintenant à ce qui nous intéresse : l'Actionscript !
Lorsque vous dessinez grâce à la classe `Graphics`, vous disposez un curseur fictif que vous pouvez déplacer n'importe où sur l'écran. Pour modifier la position de celui-ci, vous devez utiliser la méthode `moveTo()` en spécifiant le nouvel emplacement, comme ci-dessous :

Code : Actionscript

```
var _x:Number = 25;
var _y:Number = 50;
monDessin.graphics.moveTo(_x, _y);
```

Lorsque vous tracerez des lignes et des courbes, rappelez-vous que vous devez au préalable modifier la position de votre curseur. N'oubliez pas également qu'après un tracé, sa position est automatiquement mise à jour, ce qui facilite la succession de diverses lignes et courbes.

Les lignes droites

Pour réaliser des contours, il est possible de créer des lignes droites. Pour cela, la classe `Graphics` dispose de la méthode `lineTo()`, qui trace une ligne entre la position actuelle du curseur et le nouveau point spécifié en paramètre. Voici un exemple qui permet de tracer une droite horizontale d'épaisseur 10 entre les points de coordonnées (10,10) et (10,100) :

Code : Actionscript

```
monDessin.graphics.lineStyle(10, 0x000000);
monDessin.graphics.moveTo(10, 10);
monDessin.graphics.lineTo(200, 10);
```

Si vous lancez votre projet, vous verrez apparaître la belle ligne visible à la figure suivante.

Une ligne

Remarquez qu'ici nous n'avons pas utilisé les méthodes `beginFill()` et `endFill()`, nous avons donc tracé un contour sans remplissage, en utilisant uniquement un style de ligne. Notez également que nous avons d'abord déplacé notre curseur avant de dessiner la ligne.

À la suite de votre ligne horizontale, vous pourriez très bien continuer le tracé de votre contour avec une ligne verticale par exemple :

Code : Actionscript

```
monDessin.graphics.lineTo(200, 200); // Le curseur est à la
position (200,10) avant l'instruction
```

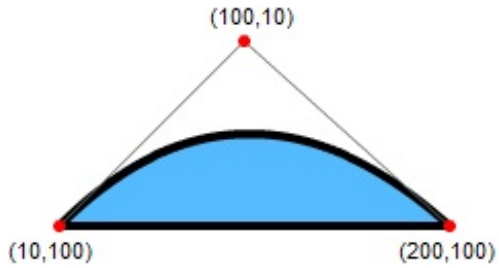
Les courbes

Les lignes droites c'est bien, mais celles-ci sont vite limitées lorsqu'il s'agit de réaliser un dessin complexe avec des formes arrondis. Par conséquent il a été mis en place ce qu'on appelle les **courbes de Bézier** !
Le principe de ces courbes est d'utiliser un **point de contrôle** en plus des deux **points d'ancrage**. Ce point de contrôle sert à définir la courbure de la ligne en direction de ce point. Pour mieux comprendre l'effet de celui-ci sur la courbe, je vous propose un exemple :

Code : Actionscript

```
monDessin.graphics.lineStyle(4, 0x000000);
monDessin.graphics.beginFill(0x55BBFF);
monDessin.graphics.moveTo(10, 100);
monDessin.graphics.curveTo(100, 10, 200, 100);
monDessin.graphics.endFill();
```

Ici, la méthode `curveTo()` prend deux paires de coordonnées en paramètres. La première correspond au point de contrôle et la seconde au point d'ancrage final. Voyez plutôt à la figure suivante le résultat du tracé où j'ai ajouté les trois points.



Une courbe décrite par trois points

Dans l'exemple précédent, nous avons définit un remplissage à l'aide de la méthode `beginFill()`. Vous constaterez alors que le contour a été automatiquement refermé par une ligne droite lors de l'appel de la méthode `endFill()`.

Les formes prédéfinis

Pour les formes géométriques particulières, il existe des méthodes permettant de simplifier leur tracé. Nous pouvons notamment citer les cercles, les ellipses et les rectangles.

Les formes elliptiques

Lorsqu'on parle de formes elliptiques, on inclut généralement les cercles et les ellipses. Nous allons donc voir à présent comment tracer chacune de ces formes.
Tout d'abord, occupons-nous des cercles. Nous disposons pour cela d'une méthode nommée `drawCircle()`, dans laquelle nous devons spécifier les coordonnées `x` et `y` du centre du cercle ainsi que le rayon `r` de celui-ci :

Code : Actionscript

```
monDessin.graphics.drawCircle(x, y, r);
```

Pour les ellipses, la méthode a utiliser est `drawEllipse()`. Celle-ci prend cette fois quatre paramètres différents : la `largeur` et la `hauteur` de l'ellipse ainsi que les coordonnées `x` et `y` du coin supérieur gauche de la zone à tracer :

Code : Actionscript

```
monDessin.graphics.drawEllipse(x, y, largeur, hauteur);
```

Nous verrons dans l'exercice à venir comment utiliser ces fonctions, mais je vous invite grandement à faire des tests de votre côté pour bien voir l'influence des différents paramètres. N'oubliez pas également de définir un style de ligne ou un remplissage avant l'utilisation de telles méthodes.

Les rectangles

En plus des formes elliptiques, la classe `Graphics` vous permet de tracer des formes rectangulaires. La méthode `drawRect()` est ainsi définie exactement de la même façon que `drawEllipse()`, à la seule différence qu'elle trace cette fois un rectangle.
L'instruction ci-dessous ne présente donc aucune surprise :

Code : Actionscript

```
monDessin.graphics.drawRect(x, y, largeur, hauteur);
```

Il se peut que vous ayez besoin, à un moment ou à un autre, de tracer un rectangle dont les angles sont arrondis. Pour cela, vous disposez également de la méthode `drawRoundRect()`, où vous devrez renseigner en plus la valeur de l'arrondi :

Code : Actionscript

```
monDessin.graphics.drawRoundRect(x, y, largeur, hauteur, arrondi);
```

La méthode `drawRoundRect()` prend en réalité un sixième paramètre si vous désirez des arrondis non uniformes en hauteur et en largeur. Ainsi le cinquième paramètre définit la largeur des arrondis et le sixième leur hauteur.

Techniques avancées

Dessiner une trajectoire

Lorsque vous créerez des contour à l'aide des méthodes `lineTo()` et `curveTo()`, il est probable que vous ayez une série d'instructions similaires afin de parcourir l'ensemble de votre contour. C'est pourquoi il existe une méthode `drawPath()` qui permet d'effectuer l'ensemble de ces tracés en une seule instruction.
Cette méthode prend donc en paramètres deux tableaux de type `Vector.<int>` et `Vector.<Number>`. Le premier permet de définir le type de commande, quant au second il contient l'ensemble des coordonnées des différents points d'ancrage et points de contrôle. En ce qui concerne les commandes, celles-ci sont définit comme suit :

- `1` → `moveTo()`
- `2` → `lineTo()`
- `3` → `curveTo()`.

Pour mieux comprendre, je vous propose de prendre l'exemple suivant :

Code : Actionscript

```
monDessin.graphics.lineStyle(2, 0x000000);
monDessin.graphics.moveTo(0, 0);
monDessin.graphics.lineTo(100, 0);
monDessin.graphics.curveTo(200, 0, 200, 100);
```

L'ensemble de ces tracés pourrait également être réalisés grâce à la méthode `drawPath()` de la façon suivante :

Code : Actionscript

```
var commandes:Vector.<int> = new Vector.<int>();
commandes.push(1, 2, 3);
var coordonnees:Vector.<Number> = new Vector.<Number>();
coordonnees.push(0, 0);
coordonnees.push(100, 0);
coordonnees.push(200, 0, 200, 100);
monDessin.graphics.lineStyle(2, 0x000000);
mouton.graphics.drawPath(commandes, coordonnees);
```

Vous remarquerez que nous aurions pu ajouter l'ensemble des coordonnées en une seule instruction avec la méthode `push()`. Néanmoins pour garder un maximum de lisibilité, je vous conseille de les ajouter par commande, comme si vous utilisiez directement les méthodes correspondantes.

N'est-ce pas plus compliqué et plus long au final ?

Je vous avoue que cette écriture fait peur et est au final plus longue à écrire que la version originale. Cependant, mettez-vous en tête que cette nouvelle version s'exécute plus rapidement que la précédente. Ceci vient du fait que le tracé à l'écran se fait en une seule fois contrairement au premier code où il faut retracer à chaque instruction. Utilisez donc cette méthode au maximum, dès que vous pouvez l'utiliser.

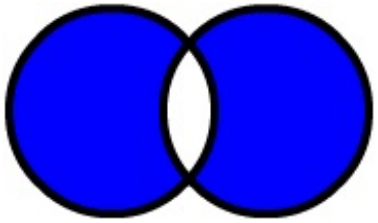
Superposition de remplissages

Lorsque vous tracez différents contours, ceux peuvent être amenés à se superposer. Dans ce cas, il est alors possible de supprimer le remplissage dans la zone de superposition. Pour cela définissez les deux contours à l'intérieur d'un même remplissage, c'est-à-dire sans fermeture du contour entre les deux par la fonction `endFill()`.
Pour vous montrer l'effet créé, voici le code que je vous invite à tester :

Code : Actionscript

```
monDessin.graphics.lineStyle(4, 0x000000);
monDessin.graphics.beginFill(0x0000FF);
monDessin.graphics.drawCircle(100, 100, 50);
monDessin.graphics.drawCircle(175, 100, 50);
monDessin.graphics.endFill();
```

Vous verrez alors deux disques de couleur bleu dont l'intersection est non remplie comme sur la figure suivante.



Exclusion des deux cercles

Un petit pas vers la 3D

Avant de terminer la théorie sur cette classe `Graphics`, je vais rapidement vous parler d'une méthode un peu spéciale : `drawTriangles()`.
Comme son nom l'indique, cette méthode permet de dessiner des triangles. Cette manière de découper une géométrie en **faces** triangulaires, est une pratique courante lorsqu'il s'agit de *3D*. Cela permet notamment de pouvoir déformer une image en la divisant sur plusieurs faces orientées différemment. Ainsi il est possible d'appliquer ce que l'on appelle une **texture**, sur un objet 3D modélisé à partir d'une multitude de faces.

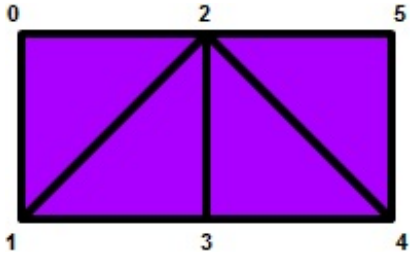
Dans ce chapitre, nous n'aborderons pas toutes ces techniques qui sont un peu trop avancées pour nous. Toutefois, je vais vous présenter tout de même comment utiliser cette méthode `drawTriangles()` avec des lignes et des remplissages. Le principe ici est de créer une liste de points que nous nommerons des **sommets**, à l'intérieur d'un tableau de type `Vector.<Number>` contenant leurs coordonnées. Ces sommets sont alors numérotés dans l'ordre de leur déclaration avec pour premier indice `0`. Un second tableau de type `Vector.<int>` permet ensuite créer des triangles en reliant les sommets par trois. Ainsi le premier triangle est généralement constitué des sommets `0`, `1` et `2`.

Conscient que tout cela est certainement encore flou dans votre tête, je vous propose un petit exemple où nous aurons 5 sommets. Ensuite, nous allons créer quatre triangles composés respectivement des sommets (0,1,2), (1,2,3), (2,3,4) et enfin (2,4,5). Voici le code correspondant à ces manipulations :

Code : Actionscript

```
var sommets:Vector.<Number> = Vector.<Number>([10, 10, 10, 100, 100,
10, 100, 100, 190, 100, 190, 10]);
var indices:Vector.<int> = Vector.<int>([0, 1, 2, 1, 2, 3, 2, 3, 4,
2, 4, 5]);
monDessin.graphics.lineStyle(4, 0x000000);
monDessin.graphics.beginFill(0xAA00FF);
monDessin.graphics.drawTriangles(sommets, indices);
monDessin.graphics.endFill();
```

Vous verrez donc apparaître nos quatre triangles formant ensemble un rectangle, qui pourrait être la base de la modélisation d'un intérieur quelconque par exemple. Voyez à la figure suivante l'affichage correspondant, où j'ai volontairement rajouté les indices des sommets pour que vous puissiez mieux comprendre :



Conception à base de triangles

Cette méthode `drawTriangles()` est principalement utilisé pour la 3D, c'est pourquoi il n'est pas nécessaire que vous compreniez son fonctionnement dans le détail. Néanmoins pour les intéressés, un troisième paramètre permet de créer de l'**UV Mapping**. Ainsi vous pouvez renseigner un vecteur `Vector.<Number>` de coordonnées *UV* pour appliquer une texture à chaque face. Pour chaque sommet de triangles, vous devez alors faire correspondre un point d'une image en coordonnées UV. Ces coordonnées UV sont alors standardisées à (0,0) pour le coin supérieur gauche de l'image bitmap et (1,1) pour le coin inférieur droit. Ne pouvant pas m'attarder plus sur le sujet, j'invite les intéressés à se rendre dans la rubrique correspondante de la documentation officielle.

Exercice : Dessine-moi un mouton

Conception du dessin

Introduction

Dans cet exercice, nous allons réaliser une petite illustration pas à pas, représentant un mouton !

Avant de nous lancer tête baissée, je vous invite à préparer votre projet afin d'obtenir un résultat similaire au mien à la fin de l'exercice. C'est pourquoi vous allez changer les dimensions de votre scène principale, pour que celle-ci fasse 280 x 220 pixels. Vous pouvez laisser la couleur d'arrière-plan en blanc étant donné que nous allons redéfinir un fond dans peu de temps. Également nous allons allons déclarer deux variables `_x` et `_y` pour contrôler la position du mouton, puis une variable `Mouton` de type `Shape` qui nous servira à effectuer l'ensemble de nos tracés. Enfin n'oubliez pas d'ajouter notre `Mouton` à notre conteneur principal grâce à la fonction `addChild()`.

Voici donc notre classe `Main` au début de l'exercice :

Code : Actionscript

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.display.Shape;

    public class Main extends Sprite
    {
        private var _x:int;
        private var _y:int;

        public function Main():void
        {
            if (stage) init();
            else addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void
        {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // entry point
            _x = 160;
            _y = 40;
            var mouton:Shape = new Shape();

            this.addChild(mouton);
        }

    }

}
```

Le fond

Afin de faire ressortir notre mouton blanc, nous allons tracer une zone rectangulaire de couleur bleu comme arrière-plan de notre dessin. Pour cela nous nous servirons de la fonction `drawRoundRect()` qui permet d'obtenir un rectangle aux coins arrondis, ce qui rendra l'ensemble plus esthétique.
Ici aucune difficulté, nous aurons donc le code suivant :

Code : Actionscript

```
// Fond
mouton.graphics.beginFill(0x10879D);
mouton.graphics.drawRoundRect(0, 0, 280, 220, 40, 40);
mouton.graphics.endFill();
```

Le corps

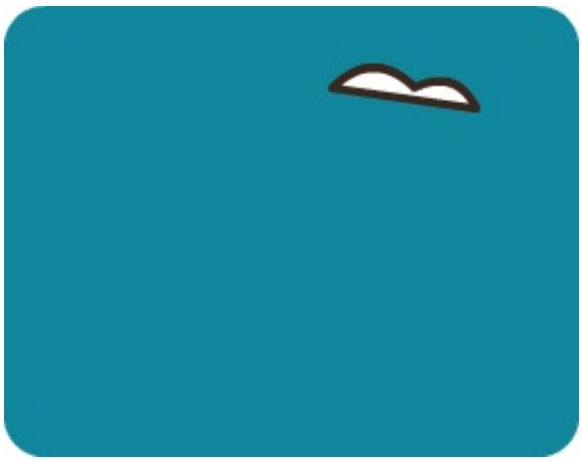
La réalisation du corps et certainement la partie la plus longue et fastidieuse de cet exercice. Néanmoins j'ai déjà effectué l'opération, vous n'aurez donc qu'à suivre mes instructions.
Pour dessiner le corps du mouton, nous allons dans un premier temps définir les différents paramètres de remplissage et de contour. Puis nous commencerons par tracer une courbe de Bézier qui sera le point de départ pour la suite. Voici donc un premier bout de code :

Code : Actionscript

```
// Corps
mouton.graphics.lineStyle(4, 0x3A281E, 1.0);
mouton.graphics.beginFill(0xFFFFFF);
mouton.graphics.moveTo(_x, _y);
mouton.graphics.curveTo(_x + 20, _y - 20, _x + 40, _y);
mouton.graphics.endFill();
```

Comme vous l'imaginez, nous ne pouvons pas dessiner l'intégralité du corps du mouton d'une seule traite. Ainsi comme vous pouvez le voir à la figure suivante, il faut procéder étape par étape :

Mise en place du premier arrondi



Création pas à pas du corps

Tracer chaque « bosse » du corps est une étape qui peut être extrêmement longue suivant vos talents en dessin. D'autre part, il est peut probable que vous obteniez rapidement un rendu proche du mien, c'est pourquoi je vous épargnerez cette tâche périlleuse. Je vous propose plutôt le code final permettant de dessiner l'intégralité du corps :

Code : Actionscript

```
// Corps
mouton.graphics.lineStyle(4, 0x3A281E, 1.0);
mouton.graphics.beginFill(0xFFFFFF);
mouton.graphics.moveTo(_x, _y);
mouton.graphics.curveTo(_x + 20, _y - 20, _x + 40, _y);
mouton.graphics.curveTo(_x + 60, _y - 10, _x + 70, _y + 10);
mouton.graphics.curveTo(_x + 90, _y + 10, _x + 90, _y + 30);
mouton.graphics.curveTo(_x + 110, _y + 40, _x + 100, _y + 60);
mouton.graphics.curveTo(_x + 120, _y + 80, _x + 100, _y + 90);
mouton.graphics.curveTo(_x + 110, _y + 110, _x + 90, _y + 110);
mouton.graphics.curveTo(_x + 80, _y + 140, _x + 60, _y + 120);
mouton.graphics.curveTo(_x + 40, _y + 150, _x + 20, _y + 130);
mouton.graphics.curveTo(_x, _y + 150, _x -20, _y + 130);
mouton.graphics.curveTo(_x -40, _y + 140, _x -50, _y + 120);
mouton.graphics.curveTo(_x -70, _y + 120, _x -70, _y + 100);
mouton.graphics.curveTo(_x -100, _y + 90, _x -90, _y + 70);
mouton.graphics.curveTo(_x -110, _y + 50, _x -90, _y + 40);
mouton.graphics.curveTo(_x -90, _y + 20, _x -60, _y + 20);
mouton.graphics.curveTo(_x -50, _y, _x -30, _y + 10);
mouton.graphics.curveTo(_x -20, _y -10, _x, _y);
mouton.graphics.endFill();
```

Voilà, vous devriez maintenant avoir un beau corps de mouton (voir figure suivante), qui je l'avoue ressemble pour l'instant plus à un nuage qu'à autre chose.

Le corps du mouton

La tête

À présent, nous allons nous occuper de la tête de notre mouton. Nous lui ferons donc deux oreilles, puis nous ajouterons la forme globale du visage. Les yeux quant à eux seront réalisés plus tard puisqu'ils demanderont un peu plus de travail.

Commençons par les oreilles, que nous créerons à partir de courbes de Bézier. La partie haute des oreilles sera dissimulée plus tard, il est donc inutile de s'attarder dessus. C'est pourquoi nous tracerons uniquement une courbe pour chaque oreille, et nous laisserons le soin à la fonction `endFill()` de refermer chacun de nos remplissages.
Ainsi le code présenté ci-dessous permet de réaliser les deux oreilles de la bête :

Code : Actionscript

```
// Tête
mouton.graphics.beginFill(0xF9D092);
mouton.graphics.moveTo(_x - 30, _y + 50);
mouton.graphics.curveTo(_x - 90, _y + 165, _x - 10, _y + 60);
mouton.graphics.endFill();
mouton.graphics.beginFill(0xF9D092);
mouton.graphics.moveTo(_x + 50, _y + 50);
mouton.graphics.curveTo(_x + 100, _y + 165, _x + 30, _y + 60);
mouton.graphics.endFill();
```

Bien évidemment, nous avons utilisé une nouvelle couleur de remplissage pour donner un effet de peau à nos oreilles que vous pouvez voir à la figure suivante.

Mise en place des oreilles

Je suggère maintenant que nous passions au visage. Pour dessiner celui-ci, nous allons commencer par tracer une ellipse. Puis nous utiliserons une ligne droite ainsi qu'une courbe de Bézier *sans remplissage* pour représenter le museau de l'animal.
Là encore il n'y a aucune difficulté, il suffit de faire les choses dans l'ordre :

Code : Actionscript

```
mouton.graphics.beginFill(0xF9D092);
mouton.graphics.drawEllipse(_x - 30, _y +20, 80, 150);
mouton.graphics.endFill();
mouton.graphics.moveTo(_x - 5, _y + 155);
mouton.graphics.curveTo(_x + 10, _y + 165, _x + 25, _y + 155);
mouton.graphics.moveTo(_x + 10, _y + 160);
mouton.graphics.lineTo(_x + 10, _y + 170);
```

Voici donc notre mouton qui commence doucement à prendre forme, comme le montre la figure suivante.

Notre mouton et sa tête

Les cheveux

Nous allons à présent dessiner ce que j'ai appelé les cheveux, mais qui correspond en réalité à la petite touffe de laine présente sur le dessus de la tête.
La technique utilisée ici est similaire à celle employée pour le corps, cependant nous allons utiliser cette fois la méthode `drawPath()`. Afin de rendre le code plus lisible, je vous conseille encore une fois d'ajouter les différentes coordonnées par commande. Voici donc le code que j'ai réalisé pour tracer les cheveux de notre beau mouton :

Code : Actionscript

```
// Cheveux
var commandes:Vector.<int> = new Vector.<int>();
commandes.push(1, 3, 3, 3, 3, 3, 3, 3, 3);
var coordonnees:Vector.<Number> = new Vector.<Number>();
coordonnees.push(_x - 20, _y + 20);
coordonnees.push(_x, _y - 10, _x + 20, _y + 10);
coordonnees.push(_x + 40, _y, _x + 50, _y + 30);
coordonnees.push(_x + 80, _y + 30, _x + 60, _y + 50);
coordonnees.push(_x + 70, _y + 70, _x + 40, _y + 70);
coordonnees.push(_x + 20, _y + 90, _x, _y + 70);
coordonnees.push(_x - 20, _y + 90, _x - 30, _y + 60);
coordonnees.push(_x - 60, _y + 50, _x - 40, _y + 30);
coordonnees.push(_x - 40, _y, _x - 20, _y + 20);
mouton.graphics.beginFill(0xFFFFFF);
mouton.graphics.drawPath(commandes, coordonnees);
mouton.graphics.endFill();
```

Nous avons cette fois quelque chose qui commence réellement à ressembler à un mouton (voir figure suivante).

Mise en place des cheveux

Les pattes

Continuons l'exercice en dessinant les pattes, dont je vous avouerai, je n'ai pas réussi à trouver l'effet que je souhaitais. Ce n'est pas grave, contentons-nous de dessiner deux rectangles arrondis en guise de pieds. Rien de plus simple, ceci est réalisable en quatre instructions :

Code : Actionscript

```
// Pattes
mouton.graphics.beginFill(0xF9D092);
mouton.graphics.drawRoundRect(_x - 60, _y + 155, 40, 20, 20, 20);
mouton.graphics.drawRoundRect(_x + 40, _y + 155, 40, 20, 20, 20);
mouton.graphics.endFill();
```

Voici donc à la figure suivante notre mouton qui peut à présent se tenir debout sur ces pattes.

Notre mouton se tient debout

Les yeux

Notre mouton est très beau, mais il est pour l'instant aveugle !
Nous allons maintenant nous occuper des yeux. Pour ceux-ci, nous n'utiliserons que des cercles, ou plutôt des disques. Il nous en faudra trois pour chaque œil : un pour l'œil en lui-même, un pour la pupille et enfin un dernier pour simuler le reflet blanc à l'intérieur de celle-ci. Un petit détail supplémentaire, nous n'aurons pas besoin de contours pour les deux derniers disques. C'est pourquoi nous définirons un style de lignes transparent pour ceux-ci.
Je vous propose donc de découvrir le code correspondant à l'ensemble de ces manipulations :

Code : Actionscript

```
// Yeux
mouton.graphics.beginFill(0xFFFFFF);
mouton.graphics.drawCircle(_x - 10, _y + 100, 15);
mouton.graphics.drawCircle(_x + 25, _y + 100, 20);
mouton.graphics.endFill();
mouton.graphics.lineStyle(1, 0x3A281E, 0.0);
mouton.graphics.beginFill(0x000000);
mouton.graphics.drawCircle(_x - 10, _y + 100, 8);
mouton.graphics.drawCircle(_x + 25, _y + 100, 8);
mouton.graphics.endFill();
mouton.graphics.beginFill(0xFFFFFF);
mouton.graphics.drawCircle(_x - 8, _y + 98, 2);
mouton.graphics.drawCircle(_x + 27, _y + 98, 2);
mouton.graphics.endFill();
```

Cette petite touche finale vient donner vie à notre mouton qui nous scrute maintenant du regard (voir figure suivante) !

Le mouton intégral

La bulle

J'aurai pu laisser ce « chef-d'œuvre » tel quel, mais je trouvais qu'il manquait un petit quelque chose. J'ai donc ajouté une bulle au mouton comme si on se trouvait dans une bande dessinée.
Rien de nouveau ici, il s'agit presque exclusivement de cercles remplis ainsi que d'un rectangle arrondi. Encore une fois, la seule difficulté est de placer les éléments au bon endroit. Voici comment faire :

Code : Actionscript

```
// Bulle
mouton.graphics.beginFill(0x3A281E);
mouton.graphics.drawCircle(_x - 60, _y + 70, 5);
mouton.graphics.drawCircle(_x - 80, _y + 50, 10);
mouton.graphics.drawRoundRect(_x - 150, _y - 25, 90, 60, 60, 60);
mouton.graphics.endFill();
mouton.graphics.beginFill(0xFFFFFF);
mouton.graphics.drawCircle(_x - 120, _y + 15, 5);
mouton.graphics.drawCircle(_x - 105, _y + 15, 5);
mouton.graphics.drawCircle(_x - 90, _y + 15, 5);
mouton.graphics.endFill();
```

Cette fois, nous avons à la figure suivante le résultat final de notre dessin.

L'illustration complète

Malgré ce que vous avez peut-être ressenti au cours de cet exercice, la réalisation de ce mouton a demandé beaucoup de travail, et je m'y suis repris à plusieurs fois pour arriver à ce résultat. N'hésitez pas à essayer de réaliser d'autres dessins plus simples pour vous faire la main avec les méthodes de cette classe `Graphics` !

Code final

Pour terminer ce chapitre, je vous ai réécrit l'intégralité du code afin que vous puissiez effectuer vos propres tests.

Code : Actionscript

```
package
{
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.display.Shape;

    public class Main extends Sprite
    {
        private var _x:int;
        private var _y:int;

        public function Main():void
        {
            if (stage) init();
            else addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void
        {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // entry point
            _x = 160;
            _y = 40;
            var mouton:Shape = new Shape();

            // Fond
            mouton.graphics.beginFill(0x10879D);
            mouton.graphics.drawRoundRect(0, 0, 280, 220, 40, 40);
            mouton.graphics.endFill();

            // Corps
            mouton.graphics.lineStyle(4, 0x3A281E, 1.0);
            mouton.graphics.beginFill(0xFFFFFF);
            mouton.graphics.moveTo(_x, _y);
            mouton.graphics.curveTo(_x + 20, _y - 20, _x + 40, _y);
            mouton.graphics.curveTo(_x + 60, _y - 10, _x + 70, _y +
10);
            mouton.graphics.curveTo(_x + 90, _y + 10, _x + 90, _y +
30);
            mouton.graphics.curveTo(_x + 110, _y + 40, _x + 100, _y
+ 60);
            mouton.graphics.curveTo(_x + 120, _y + 80, _x + 100, _y
+ 90);
            mouton.graphics.curveTo(_x + 110, _y + 110, _x + 90, _y
+ 110);
            mouton.graphics.curveTo(_x + 80, _y + 140, _x + 60, _y +
120);
            mouton.graphics.curveTo(_x + 40, _y + 150, _x + 20, _y +
130);
            mouton.graphics.curveTo(_x, _y + 150, _x -20, _y + 130);
            mouton.graphics.curveTo(_x -40, _y + 140, _x -50, _y +
120);
            mouton.graphics.curveTo(_x -70, _y + 120, _x -70, _y +
100);
            mouton.graphics.curveTo(_x -100, _y + 90, _x -90, _y +
70);
            mouton.graphics.curveTo(_x -110, _y + 50, _x -90, _y +
40);
            mouton.graphics.curveTo(_x -90, _y + 20, _x -60, _y +
20);
            mouton.graphics.curveTo(_x -50, _y, _x -30, _y + 10);
            mouton.graphics.curveTo(_x -20, _y -10, _x, _y);
            mouton.graphics.endFill();

            // Tête
            mouton.graphics.beginFill(0xF9D092);
```

```
                mouton.graphics.moveTo(_x - 30, _y + 50);
                mouton.graphics.curveTo(_x - 90, _y + 165, _x - 10, _y +
60);
                mouton.graphics.endFill();
                mouton.graphics.beginFill(0xF9D092);
                mouton.graphics.moveTo(_x + 50, _y + 50);
                mouton.graphics.curveTo(_x + 100, _y + 165, _x + 30, _y
+ 60);
                mouton.graphics.endFill();
                mouton.graphics.beginFill(0xF9D092);
                mouton.graphics.drawEllipse(_x - 30, _y +20, 80, 150);
                mouton.graphics.endFill();
                mouton.graphics.moveTo(_x - 5, _y + 155);
                mouton.graphics.curveTo(_x + 10, _y + 165, _x + 25, _y +
155);
                mouton.graphics.moveTo(_x + 10, _y + 160);
                mouton.graphics.lineTo(_x + 10, _y + 170);

                // Cheveux
                var commandes:Vector.<int> = new Vector.<int>();
                commandes.push(1, 3, 3, 3, 3, 3, 3, 3, 3);
                var coordonnees:Vector.<Number> = new Vector.<Number>();
                coordonnees.push(_x - 20, _y + 20);
                coordonnees.push(_x, _y - 10, _x + 20, _y + 10);
                coordonnees.push(_x + 40, _y, _x + 50, _y + 30);
                coordonnees.push(_x + 80, _y + 30, _x + 60, _y + 50);
                coordonnees.push(_x + 70, _y + 70, _x + 40, _y + 70);
                coordonnees.push(_x + 20, _y + 90, _x, _y + 70);
                coordonnees.push(_x - 20, _y + 90, _x - 30, _y + 60);
                coordonnees.push(_x - 60, _y + 50, _x - 40, _y + 30);
                coordonnees.push(_x - 40, _y, _x - 20, _y + 20);
                mouton.graphics.beginFill(0xFFFFFF);
                mouton.graphics.drawPath(commandes, coordonnees);
                mouton.graphics.endFill();

                // Pattes
                mouton.graphics.beginFill(0xF9D092);
                mouton.graphics.drawRoundRect(_x - 60, _y + 155, 40, 20,
20, 20);
                mouton.graphics.drawRoundRect(_x + 40, _y + 155, 40, 20,
20, 20);
                mouton.graphics.endFill();

                // Yeux
                mouton.graphics.beginFill(0xFFFFFF);
                mouton.graphics.drawCircle(_x - 10, _y + 100, 15);
                mouton.graphics.drawCircle(_x + 25, _y + 100, 20);
                mouton.graphics.endFill();
                mouton.graphics.lineStyle(1, 0x3A281E, 0.0);
                mouton.graphics.beginFill(0x000000);
                mouton.graphics.drawCircle(_x - 10, _y + 100, 8);
                mouton.graphics.drawCircle(_x + 25, _y + 100, 8);
                mouton.graphics.endFill();
                mouton.graphics.beginFill(0xFFFFFF);
                mouton.graphics.drawCircle(_x - 8, _y + 98, 2);
                mouton.graphics.drawCircle(_x + 27, _y + 98, 2);
                mouton.graphics.endFill();

                // Bulle
                mouton.graphics.beginFill(0x3A281E);
                mouton.graphics.drawCircle(_x - 60, _y + 70, 5);
                mouton.graphics.drawCircle(_x - 80, _y + 50, 10);
                mouton.graphics.drawRoundRect(_x - 150, _y - 25, 90, 60,
60, 60);
                mouton.graphics.endFill();
                mouton.graphics.beginFill(0xFFFFFF);
                mouton.graphics.drawCircle(_x - 120, _y + 15, 5);
                mouton.graphics.drawCircle(_x - 105, _y + 15, 5);
                mouton.graphics.drawCircle(_x - 90, _y + 15, 5);
                mouton.graphics.endFill();
```

```
                this.addChild(mouton);
            }

        }

    }
```

En résumé

- La classe `Graphics` renferme toutes les méthodes permettant de dessiner directement depuis le code.
- Pour accéder à cette classe `Graphics`, il est nécessaire de passer par l'instanciation de l'une de ces trois classes : `Shape`, `Sprite` ou `MovieClip`.
- En Flash, les dessins réalisés sont de type **vectoriels**, qui s'adaptent à la taille de l'affichage.
- Pour dessiner en Actionscript, nous devons définir un **contour** puis spécifier les styles de **remplissages** et de **lignes** qui doivent lui être appliqués.
- Les contours peuvent être réalisés grâce à des lignes droites ou des **courbes de Bézier**.
- Des formes prédéfinies facilitent le tracé de formes géométriques simples, telles que les cercles, les ellipses et les rectangles.

Utilisation des matrices

Nous allons à présent nous pencher sur les matrices ! N'ayez pas peur, ce chapitre ne sera pas aussi complexe qu'il en à l'air. En fait, l'utilisation de ces matrices est grandement facilitée en Actionscript. En effet, vous verrez que nous disposons d'une classe nommée `Matrix` qui permet de manipuler une matrice sans besoin de comprendre comment elle fonctionne réellement. Là encore, je pense que nous pouvons remercier la POO et l'encapsulation.

Nous verrons donc comment utiliser cette classe pour décrire comment appliquer un dégradé dans un tracé, ou encore effectuer des transformations géométriques à nos objets d'affichage.

Les matrices ou la classe Matrix

Introduction aux matrices

Les bases

Avant d'aller plus loin, je pense qu'il serait bien de vous expliquer un peu ce qu'est une matrice, si vous ne le savez pas déjà. Ne vous inquiétez pas, il ne s'agit pas d'un cours de mathématiques. Nous ne verrons donc pas de longues formules où vous risqueriez de vous arracher les cheveux !

Malgré son nom barbare, une matrice n'est en réalité rien d'autre qu'un tableau de valeurs. Les nombres qui sont stockés peuvent représenter tout un tas de choses, suivant l'utilisation que l'on peut en avoir. Les matrices peuvent donc être utilisées pour des applications simples, mais permettent également de représenter des concepts mathématiques complexes. Toutefois en ce qui nous concerne, nous nous contenterons de dire qu'une matrice est un tableau de nombres et nous nous limiterons aux matrices bidimensionnelles.
Ainsi toute matrice est de taille $m \times n$ où m est le nombre de lignes et n le nombre de colonnes. Voici par exemple une matrice 3×3 quelconque :

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Comme dans un tableau, il est alors possible de désigner un élément particulier grâce à un couple d'indices noté (i, j). Dans l'exemple précédent, supposons que nous nommions la matrice A. Nous pourrions par exemple préciser que $A(2, 1) = 4$.
En mathématiques, il est possible de réaliser tout un tas d'opérations avec les matrices. Cependant ce n'est pas ici l'objectif de vous présenter tout cela, qui serait bien trop long et n'aurait pas vraiment d'utilité dans notre cas. Toutefois, retenez qu'il existe une matrice un peu particulière où les coefficients diagonaux sont égaux à 1 et les autres coefficients sont tous nuls : la **matrice identité**. En voici une :

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Pour les débutants en la matière, vous retiendrez donc qu'une matrice peut globalement se manipuler comme un tableau de nombres.

Théorie pour les initiés

Malgré le titre, j'invite également les novices en termes de matrices à lire la suite ne serait-ce que pour le plaisir ! 😄
En deux dimensions, nous avons l'habitude d'utiliser un couple de coordonnées noté (x, y) pour repérer un point sur le plan. Lorsqu'on utilise la notation matricielle, il est alors préférable de noter ces coordonnées en vertical et nous appellerons cette nouvelle notation un **vecteur**. Voici donc notre vecteur :

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

Une des propriétés des matrices est la multiplication ; elle permet ainsi de multiplier deux matrices entre elles mais également une matrice avec un vecteur. Toutefois les dimensions des deux entités de l'opération doivent être cohérentes. Dans notre cas, pour pouvoir multiplier notre vecteur de dimension 2×1, nous devrons utiliser une matrice 2×2. L'intérêt de tout ça est qu'à l'intérieur de notre matrice 2×2, nous pouvons définir différentes transformations géométriques :

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

L'exemple ci-dessus est opération de mise à l'échelle où s_x correspond au redimensionnement horizontal et s_y au vertical. Ainsi après multiplication, nous obtenons un nouveau vecteur où les nouvelles coordonnées ont été modifiées suivant la transformation.
Une opération qui, sans doute, vous fera certainement peur est la rotation, où nous retrouvons nos fonctions préférés : *sinus* et *cosinus*. Voici donc comment se réalise une rotation d'un angle θ :

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix}$$

Nous arrivons maintenant au petit point qui a nécessité tout ce qui précède. Lorsqu'on utilise une matrice de dimension 2×2, il est possible de réaliser beaucoup de transformations. Toutefois, la translation ne peut pas être prise en compte dans ces conditions. C'est pourquoi il est impératif d'ajouter une dimension supplémentaire à l'ensemble de l'opération. Il devient alors possible de réaliser une translation de la manière suivante :

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Dans l'exemple précédent, les valeurs t_x et t_y représentent les déplacements respectivement suivant les axes $\overrightarrow{x}$ et $\overrightarrow{y}$.

Le fait de rajouter une dimension au vecteur de coordonnées n'a de sens que pour l'opération mathématique. Ainsi les coordonnées 2D du point considéré seront toujours les deux premières, la troisième étant tout simplement ignorée.

Vous noterez que toute cette théorie sur les matrices ne sert qu'à vous aider à mieux cerner l'intérêt des matrices. Dans la suite, vous verrez que toutes ces opérations sont simplifiées grâce à l'*encapsulation* de la classe `Matrix`, qui dispose de méthodes simples pour définir ces transformations. Ce n'est donc pas dramatique pour la suite si vous n'avez pas tout compris sur la théorie des matrices.

L'objet `Matrix`

Suite à cette petite introduction aux matrices, vous devriez maintenant être plus à l'aise et surtout vous devriez pas être surpris par tout ce qui va suivre.

Introduction

Comme vous l'imaginez maintenant, la classe `Matrix` permet de définir une matrice à l'intérieur du code. Celle-ci permettra donc de définir l'ensemble des transformations à appliquer à un objet.
Si vous avez bien suivi tout ce qui a été dit sur les matrices précédemment, vous devriez donc pas être surpris par la forme des matrices manipulées par cette classe :

$$\begin{bmatrix} a & c & t_x \\ b & d & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Nous retrouvons donc quatre coefficients a, b, c et d qui permettent de réaliser les transformations classiques telles que le redimensionnement, la rotation mais également l'inclinaison. Également nous disposons des valeurs de translation t_x et t_y, permettant de déplacer un objet.
Cette classe `Matrix` va donc nous permettre de définir une matrice qui combinera une ou plusieurs de ces transformations en même temps.

Déclaration et initialisation

Comme toute classe, un objet `Matrix` s'initiale grâce au constructeur de la classe dans une variable.
Voici donc comment procéder :

Code : Actionscript

```
var maTransformation:Matrix = new Matrix();
```

Lorsque vous initialisez une matrice comme ceci, le constructeur crée une matrice identité, qui en termes de transformations ne fait strictement rien.
En fait, ce constructeur possède les différents paramètres facultatifs suivant : `a`, `b`, `c`, `d`, `tx` et `ty`. Toutefois, comme nous ne voulons pas nous prendre la tête avec les mathématiques, nous utiliserons plutôt les méthodes de la classe qui simplifient énormément la gestion des transformations.

Création de dégradés

Dans le chapitre précédent, j'ai volontairement omis de vous parler des dégradés. Nous verrons dans la suite que l'utilisation des matrices permet de mieux contrôler le dégradé en lui-même, et ainsi parvenir plus facilement au rendu désiré.

Présentation des dégradés

La gestion des dégradés en Actionscript peut paraître complexe au premiers abords, cependant les possibilités en sont d'autant plus larges.
Vous verrez au fil des explications que la création d'un dégradé se fait par étapes, de la même manière qu'on le ferait dans un logiciel de graphisme. Nous allons donc dans un premier temps définir le dégradé à l'aide de différents tableaux `Array`, comme ceci :

Code : Actionscript

```
var couleurs:Array = [0xFF0000, 0x00FF00];
var alphas:Array = [1, 1];
var ratios:Array = [0, 255];
```

Dans l'exemple précédent, nous avons déclaré et initialisé trois tableaux : `couleurs`, `alphas` et `ratios`. Comme son nom l'indique, le premier sert à lister les différentes couleurs du dégradé, ici rouge et vert. Le deuxième permet de définir l'opacité de chacune des couleurs précédentes *dans le même ordre*. Enfin la variable `ratios` nous permet de gérer la position des couleurs les unes par rapport aux autres. Pour comparer, vous pouvez imaginer qu'il s'agit des petits curseurs que vous pourriez déplacer dans un éditeur de dégradé, ici représentés par des valeurs comprises entre 0 l'extrémité gauche et 255 l'extrémité droite, comme le montre la figure suivante.


Un dégradé et des curseurs

Un des intérêts de l'utilisation de tableaux vient du fait que leur taille peut être quelconque. Ainsi vous pouvez rajouter autant de couleurs que vous souhaitez, en rajoutant autant valeurs dans chacun des trois tableaux précédents. C'est principalement en ajoutant de nouvelles couleurs que la variable `ratios` prend de l'importance et permet alors de bien situer les couleurs entre elles.

En théorie, il est possible d'appliquer un dégradé tel quel à un remplissage par exemple. Nous reviendrons dessus plus tard, mais vous pourriez très bien utiliser le code suivant :

Code : Actionscript

```
var monDessin:Shape = new Shape;
monDessin.graphics.beginGradientFill(GradientType.LINEAR, couleurs,
alphas, ratios);
monDessin.graphics.drawRect(0, 0, 100, 100);
addChild(monDessin);
```

Néanmoins, il est quasiment indispensable de se servir des matrices pour décrire comment les dégradés doivent être appliqués.

Hormis les valeurs `GradientType.LINEAR` et `GradientType.RADIAL` qui permettent de définir le dégradé de type linéaire ou radial, il est nécessaire de préciser la largeur, la hauteur ou encore l'angle de ce dernier à l'intérieur du futur remplissage.
Ainsi un objet de type `Matrix` va permettre de mettre en place ces différents paramètres à l'intérieur d'une même matrice.

Ajouter une matrice de description

Pré-requis

Vous noterez que l'utilisation des matrices ici n'est tout à fait identique aux transformations décrites auparavant. Toutefois vous verrez qu'au final la description des dégradé y est très similaire et que les matrices seront alors définis de la même manière.

Dans la suite, nous partirons du dégradé blanc et noir défini ci-dessous :

Code : Actionscript

```
var couleurs:Array = [0xFFFFFF, 0x000000];
var alphas:Array = [1, 1];
var ratios:Array = [0, 255];
```

Préparation de la matrice

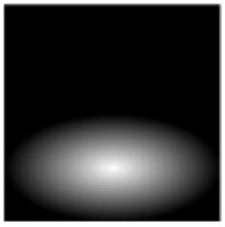
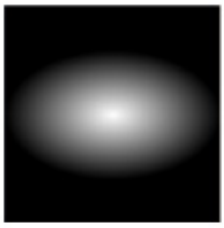
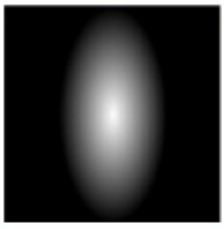
Comme nous l'avons dit, la classe `Matrix` possède des méthodes qui facilitent la définition d'une matrice. Parmi elles, la méthode `createGradientBox()` est particulièrement utile pour décrire la mise en place d'un dégradé.
Les différents paramètres de cette méthode sont listés dans le code d'exemple ci-dessous :

Code : Actionscript

```
var matrix:Matrix = new Matrix();
var largeur:Number = 100;
var hauteur:Number = 100;
var rotation:Number = 0;
var tx:Number = 0;
var ty:Number = 0;
matrix.createGradientBox(largeur, hauteur, rotation, tx, ty);
```

Notez que seuls la largeur et la hauteur sont des paramètres obligatoires. Vous pouvez donc aisément vous passer des derniers paramètres si ceux-ci sont nuls, et ainsi alléger votre code. Ce sera notamment souvent le cas pour les valeurs de décalage du dégradé `tx` et `ty`.

Le nom des variables dans l'exemple précédent est assez explicite, cependant je vous propose à la figure suivante un tableau montrant l'influence de ces différents paramètres sur le remplissage final.

Description de la matrice	Aperçu en linéaire	Aperçu en radial
largeur = 100 hauteur = 100 rotation = 0 tx = 0 ty = 0		
largeur = 100 hauteur = 50 rotation = - Math.PI/2 tx = 0 ty = 50		
largeur = 100 hauteur = 60 rotation = Math.PI/4 tx = 0 ty = 20		
largeur = 50 hauteur = 100 rotation = Math.PI/6 tx = 25 ty = 0		

Les paramètres modifient l'affichage

Comme c'est souvent le cas en programmation, les angles sont exprimés en *radians*. Toutefois l'utilisation de la classe `Math` vous permet de récupérer la valeur de π et donc d'exprimer facilement les angles dans cette unité. Ainsi la conversion en radians se fait à l'aide du facteur $\frac{\pi}{180}$.

Appliquer un dégradé

Pour appliquer un dégradé à un remplissage, vous devez utiliser la méthode `beginGradientFill()` que nous avions rapidement vu précédemment. Vous pouvez alors définir le remplissage en renseignant les différents paramètres dans l'ordre suivant :

Code : Actionscript

```
monDessin.graphics.beginGradientFill(GradientType.LINEAR, couleurs,
alphas, ratios, matrix);
```

Je rappelle qu'un dégradé peut être linéaire ou radial, défini par les constantes `GradientType.LINEAR` et `GradientType.RADIAL`. Comme pour la méthode `beginFill()`, le remplissage est terminé lors de l'appel de la fonction `endFill()` qui termine le tracé d'un contour.

Contrairement à ce que vous pourriez croire, un dégradé peut tout autant s'appliquer aux lignes. Pour cela utilisez la méthode `lineGradientStyle()` qui s'utilise exactement de la même manière que `beginGradientFill()`. Toutefois n'oubliez pas de définir la largeur de la ligne à l'aide de la méthode `lineStyle()`, sans quoi celle-ci ne sera pas visible.
Voici donc comment procéder :

Code : Actionscript

```
monDessin.graphics.lineStyle(2);
monDessin.graphics.lineGradientStyle(GradientType.LINEAR, couleurs,
alphas, ratios, matrix);
```

Exemple : création d'un bouton

Pour terminer les explications sur les dégradés, je vous propose de réaliser un petit bouton comme vous pourriez en trouver sur n'importe quelle page web.
Pour ce faire, nous allons donc utiliser un dégradé linéaire vertical que nous utiliserons comme remplissage. Nous dessinerons alors un rectangles aux coins arrondis avec une ligne de couleur pour marquer le contour de notre bouton.
Sans plus attendre, voici le code correspondant à la définition du dégradé et au tracé de ce bouton :

Code : Actionscript

```
// Dégradé de base
var couleurs:Array = [0xEEBBBB, 0xDD0000];
var alphas:Array = [1, 1];
var ratios:Array = [0, 255];

// Matrice de description
var matrix:Matrix = new Matrix();
var largeur:Number = 100;
var hauteur:Number = 15;
var rotation:Number = Math.PI/2;
var tx:Number = 0;
var ty:Number = 0;
matrix.createGradientBox(largeur, hauteur, rotation, tx, ty);

// Tracé du bouton
var monBouton:Shape = new Shape;
monBouton.graphics.lineStyle(1, 0x880000);
monBouton.graphics.beginGradientFill(GradientType.LINEAR, couleurs,
alphas, ratios, matrix);
monBouton.graphics.drawRoundRect(0, 0, 100, 25, 10);
addChild(monBouton);
```

Nous avons donc ici réalisé la base d'un bouton typique d'une page internet. Il manque encore le texte par-dessus celui-ci mais vous devriez normalement pouvoir vous en occuper vous-mêmes. La figure suivante montre à quoi ressemble notre bouton.

 Notre bouton

Pour ajouter du texte grâce à l'objet `TextField`, vous devrez utiliser un conteneur de type `Sprite` pour réaliser ce bouton. Vous pourrez alors insérer l'objet `Shape` à l'intérieur où dessiner directement depuis la propriété `graphics` du conteneur.

Les transformations matricielles

Un objet à transformer

Création de l'objet

Avant quoi que ce soit, nous allons créer une forme que nous pourrons déformer plus tard. Pour ne pas se compliquer la vie, nous tracerons donc un carré dans un objet de type `Shape`. Nous partirons donc du code ci-dessous pour la suite de ce chapitre :

Code : Actionscript

```
var monObjet:Shape = new Shape();
monObjet.graphics.beginFill(0x6688FF);
monObjet.graphics.drawRect(0, 0, 100, 100);
this.addChild(monObjet);
```

Pour que vous puissiez mieux comparer les effets des différentes transformations, voici à la figure suivante l'original que nous venons de créer.

Notre image originale

Lorsque vous créez n'importe quel objet d'affichage, il lui est associé un centre de transformation. Par défaut, le centre d'un objet est placé au point (0,0). Veillez donc à en prendre compte lors de la création de vos affichages car ce point influera sur la transformation, et il est probable que vous ayez alors besoin de replacer vos objets après chacune d'elles.

La matrice de transformation

Tout objet héritant de la classe `DisplayObject` possède une matrice de transformation. Celle-ci n'a aucune incidence sur l'affichage s'il s'agit d'une matrice identité ou si sa valeur est **`null`**. Pour y accéder, vous devez atteindre la propriété `matrix` d'une propriété de votre objet nommée `transform`.
Voici donc comment y parvenir :

Code : Actionscript

```
monObjet.transform.matrix = matrix;
```

Notez que la propriété de type `Transform` possède également une matrice de type `Matrix3D` qui permet de gérer les transformations en trois dimensions. Nous n'en parlerons pas ici, mais sachez que vous ne pouvez utiliser ces deux matrices simultanément. La modification de l'une des deux entraîne forcément l'autre à une valeur **`null`**.

Bien qu'il soit possible de modifier directement les valeurs à l'intérieur d'une matrice, nous préférerons utiliser les méthodes de la classe `Matrix` pour effectuer nos transformations.

Création d'une matrice de transformation

Méthodes de transformations individuelles

Pour commencer, nous allons voir revenir à tout moment à une matrice identité. Cela correspond donc à une matrice n'appliquant aucune transformation à l'objet. Pour cela, utilisez la fonction `identity()` simplement comme ceci :

Code : Actionscript

```
matrix.identity();
```

Ensuite, voyons comment redimensionner notre objet d'affichage. Pour cela nous utiliserons la méthode `scale()`, où nous pouvons spécifier les valeurs de redimensionnement horizontale `sx` et verticale `sx`, la taille d'origine étant fixée à `1`. Voici donc comment procéder :

Code : Actionscript

```
matrix.scale(sx, sy);
```

Je vais maintenant vous présenter la méthode `rotate()` permettant de faire pivoter un objet par rapport à son centre. il suffit alors uniquement de préciser l'angle de rotation en *radians* :

Code : Actionscript

```
matrix.rotate(angle);
```

Voyons à présent ensemble la translation. Rien de plus simple, il suffit de préciser les décalages `dx` et `dy` à la méthode `translate()` de la façon suivante :

Code : Actionscript

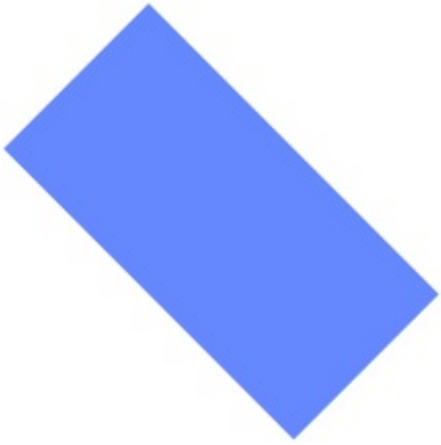
```
matrix.translate(dx, dy);
```

Enfin pour terminer, je vous propose une série de transformations dans l'exemple suivant :

Code : Actionscript

```
var matrix:Matrix = new Matrix();
matrix.identity();
matrix.scale(2, 1);
matrix.rotate(Math.PI / 4);
matrix.translate(72, 0);
// Application des transformations
monObjet.transform.matrix = matrix;
```

Dans l'exemple précédent, nous avons donc doublé notre objet dans le sens horizontal. Puis nous l'avons fait tourner de 45° par rapport à son centre confondu au coin supérieur gauche du rectangle. Enfin, nous avons translaté l'objet afin qu'il soit à nouveau entièrement dans la zone visible de l'écran. Regardez le résultat final à la figure suivante.



Le résultat final

N'oubliez pas que lorsque vous effectuez des transformations, celles-ci sont toujours réalisées par rapport au centre de votre objet. Ainsi si celui-ci n'est positionné en (0,0), vous verrez sans nul doute votre objet de déplacer également en rapport des transformations que vous pourrez lui appliquer. Soyez donc prévoyant au moment de la création de objet

pour essayer d'anticiper ces repositionnements.

Une petite exception : l'inclinaison

Il est aussi possible de déformer réellement votre objet en l'inclinant. Toutefois, la classe `Matrix` ne possède pas de méthode pour ce genre de transformation. Cependant, il est possible de modifier directement les valeurs à l'intérieur de la matrice. C'est donc ce que nous ferons pour réaliser ceci, en agissant sur la valeur b pour une inclinaison verticale et sur la c pour une horizontale.
Voici donc comment faire :

Code : Actionscript

```
matrix.c = - Math.tan(2);
```

Dans cet exemple, nous inclinons notre objet d'une valeur de `2` horizontalement. Ainsi, chaque pixel va être décalé vers la droite de deux fois sa distance verticale au centre de l'objet. Voyez plutôt le résultat :

Le résultat final

Plusieurs transformations à la fois

Contrairement à ce que pourrait vous laissez croire le titre, les méthodes précédentes permettent également de combiner plusieurs transformations en une seule matrice. Ce que nous allons faire ici, c'est créer une matrice combinant plusieurs transformations à l'aide d'une unique méthode : `createBox()`.

Cette méthode permet de combiner un redimensionnement avec une rotation et une translation. Néanmoins, il faut être vigilent quant à l'ordre d'application de chacune de ces transformations. L'ordre des opérations est le suivant : identité -> rotation -> redimensionnement -> translation.

Ainsi lorsque vous utiliserez cette méthode, la matrice sera « réinitialée » et effacera donc toute transformation de la matrice.
Voici donc comment s'utilise cette méthode :

Code : Actionscript

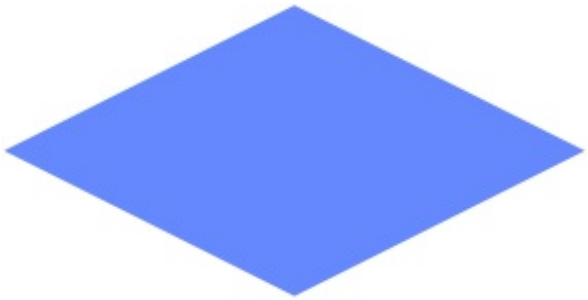
```
matrix.createBox(sx, sx, angle, dx, dy);
```

Je vous propose donc un petit exemple d'utilisation :

Code : Actionscript

```
var matrix:Matrix = new Matrix();
matrix.createBox(2, 1, Math.PI/4, 150, 0);
monObjet.transform.matrix = matrix;
```

Les opérations sont donc insérées en une seule fois à l'intérieur de la matrice, et produisent le la figure suivante.


Le résultat final

Remarquez qu'ici les transformations n'ont pas été effectuées dans le même ordre que précédemment, notamment la rotation est réalisée avant le redimensionnement. Il est donc normal que notre objet est été déformé et qu'il ressemble maintenant plus à un losange qu'à un carré.

Pour finir avec les matrices

Ajouter des transformations

Depuis le départ, nous ne faisons qu'affecter une matrice de transformation à un objet. Pour faire ça, nous utilisons donc l'instruction suivante :

Code : Actionscript

```
monObjet.transform.matrix = matrix;
```

Cependant en effectuant cette affectation, vous effacez également d'éventuelles transformations déjà appliquées à l'objet. Or vous souhaiteriez peut-être conserver ces anciennes transformations, et appliquer en plus les nouvelles. Pour faire cela, vous aurez donc besoin de récupérer la matrice existante, avant d'y ajouter de nouvelles opérations de transformation.
Voilà donc comment nous pourrions, par exemple, rajouter une rotation de 45° à notre objet, sans réinitialiser l'ensemble des transformations déjà existantes :

Code : Actionscript

```
// Récupération de la matrice de l'objet
var matrix:Matrix = new Matrix();
matrix = monObjet.transform.matrix;
// Ajout d'une transformation
matrix.rotate(Math.PI / 4);
// Application des transformations
monObjet.transform.matrix = matrix;
```

Si vous avez bien suivi, vous devriez vous rappeler qu'en utilisant la méthode `createBox()`, vous commencez par redéfinir l'objet `Matrix` comme matrice identité, éliminant ainsi toute trace d'anciennes transformations. Il faut donc procéder différemment si vous souhaitez conserver ces transformations. Pour cela, la classe `Matrix` dispose d'une méthode nommée `concat()` qui permet de concaténer une matrice avec une autre, c'est-à-dire de combiner les effets géométriques des deux matrices.
Voilà donc comment je vous suggère de procéder :

Code : Actionscript

```
// Récupération de la matrice de l'objet
var matrix:Matrix = new Matrix();
matrix = monObjet.transform.matrix;
// Création de la matrice de transformation
var transformation:Matrix = new Matrix();
transformation.createBox(2, 1, Math.PI/4, 150, 0);
```

```
// Combinaison des matrices
matrix.concat(transformation);
// Application des transformations
monObjet.transform.matrix = matrix;
```

Soyez attentif toutefois à la manière de concaténer vos deux matrices. En effet, il faut bien ici ajouter les modifications de la matrice `transformation` à la matrice d'origine `matrix`. C'est pourquoi nous utilisons la méthode `concat()` de `matrix` et non celle de `transformation`, qui aurait pour effet d'appliquer les nouvelles transformations avant les anciennes.

Intérêt des transformations matricielles

Il est probable que vous vous demandiez depuis le début quel est l'intérêt d'utiliser les matrices alors que les propriétés `x`, `y`, `rotation`, `scaleX` et `scaleY` permettent de faire la même chose.

En réalité les transformations matricielles sont plus puissantes, notamment car elles permettent les déformations impossibles avec les propriétés précédentes. Un autre avantage est également de limiter les modifications liées à l'affichage. En effet lorsque vous modifier les propriétés d'un objet d'affichage, celui-ci doit être entièrement retracé, c'est-à-dire pixel par pixel. Cette opération peut donc être longue suivant les dimensions de vos tracés et autres objets d'affichage. C'est pourquoi utiliser les transformations matricielles accélère cette étape en combinant plusieurs opérations en *une seule* instruction de mise à jour de l'affichage.
Ainsi il est préférable d'utiliser cette technique pour diminuer le temps d'exécution de ces transformations et fluidifier au maximum votre affichage.

En résumé

- Une **matrice** est un tableau de nombres à m lignes et n colonnes.
- La classe `Matrix` sert à créer et manipuler des matrices plus simplement.
- Cette classe permet notamment de faciliter la mise en place des dégradés grâce à sa méthode `createGradientBox()`.
- L'objet `Matrix` est beaucoup utilisé pour effectuer des **transformations matricielles**.
- Pour les transformations classiques, il existe les différentes méthodes `scale()`, `rotate()` et `translate()`.
- Il est possible de combiner plusieurs opérations géométriques grâce à la méthode `createBox()`
- L'utilisation des matrices pour les transformations géométriques permet d'accélérer le rendu de l'affichage.

Manipuler des images

Il est temps d'embellir notre application à l'aide d'images ! Dans ce chapitre, nous verrons comment incorporer des images à notre application, comment les afficher sur la scène, en plus de quelques manipulations utiles. Toutefois, nous n'aborderons pas le chargement dynamique d'images dans ce chapitre, mais plutôt dans la cinquième partie sur le multimédia.

Embarquer des images

Préparation de l'image

Dans un premier temps, il faut préparer l'image que nous voulons incorporer dans notre application. Reprenons l'image de Zozor (voir figure suivante) que nous avons utilisée pour afficher une image dans un champ de texte.

Zozor, la mascotte du Site du Zéro

Créez un dossier `lib` s'il n'existe pas déjà dans le répertoire de votre projet, puis créez-y un dossier `img`. Enfin, copiez dans ce dossier `img` l'image de Zozor. Voilà ! Notre image de Zozor est prête à être incorporée dans notre application !

Librairie d'images

Une nouvelle classe

Pour que notre projet soit un peu organisé, nous allons embarquer nos image dans une classe qui représentera notre bibliothèque d'images.
Créez une nouvelle classe `ImageLibrary` à côté de notre classe `Main` :

Code : Actionscript

```
package
{
    /**
* Librairie contenant nos images embarquées.
* @author Guillaume CHAU
*/
    public class ImageLibrary
    {

    }

}
```

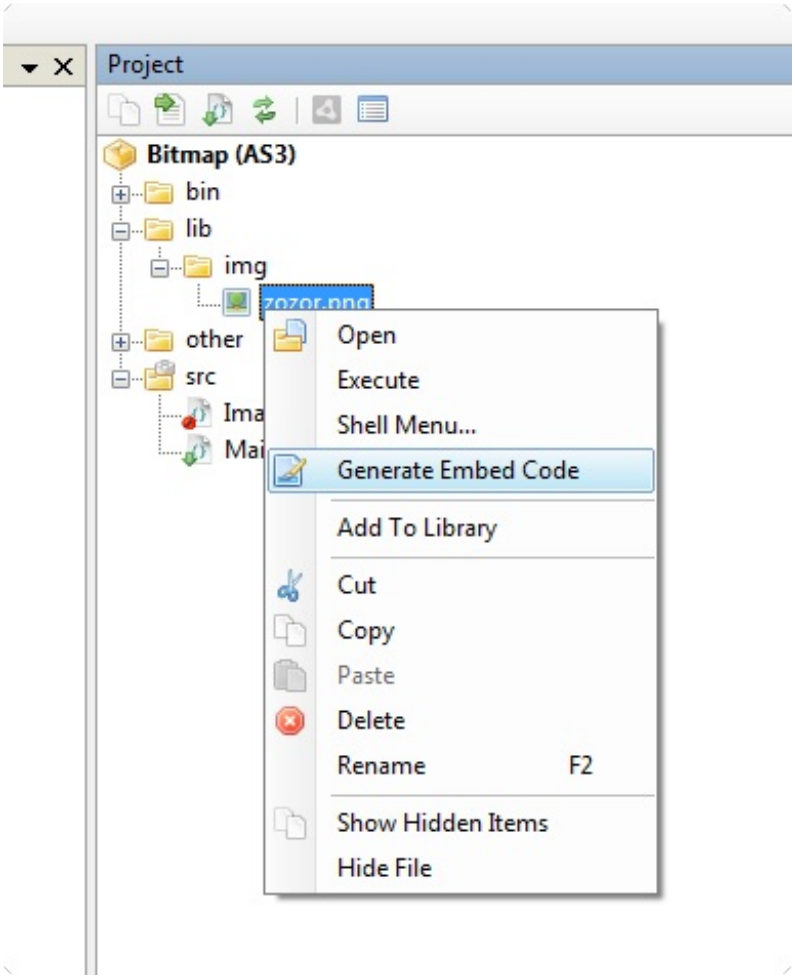
Incorporer une image

Pour embarquer une image, nous allons utiliser une instruction spéciale pour le compilateur, de la même manière que nous avons embarqué une police de caractère, grâce au mot-clé `Embed` :

Code : Actionscript

```
[Embed(source="../lib/img/zozor.png")]
```

Le code est plus simple : il suffit d'indiquer le chemin de l'image dans le paramètre `source`.
Petite astuce si vous utilisez *FlashDevelop* : dans l'arborescence du projet, faite un clic droit sur l'image à incorporer, puis sélectionnez `Generate Embed Code`, comme à la figure suivante.



FlashDevelop facilite l'incorporation d'images

FlashDevelop insère alors l'instruction spéciale automatiquement, à l'endroit où votre curseur d'insertion se trouve dans la classe.

Puis, sur la ligne suivante, il faut déclarer un attribut statique de type `Class`, comme pour les polices de caractères embarquées :

Code : Actionscript

```
public static var Zozor:Class;
```

Voici le code complet de la classe :

Code : Actionscript

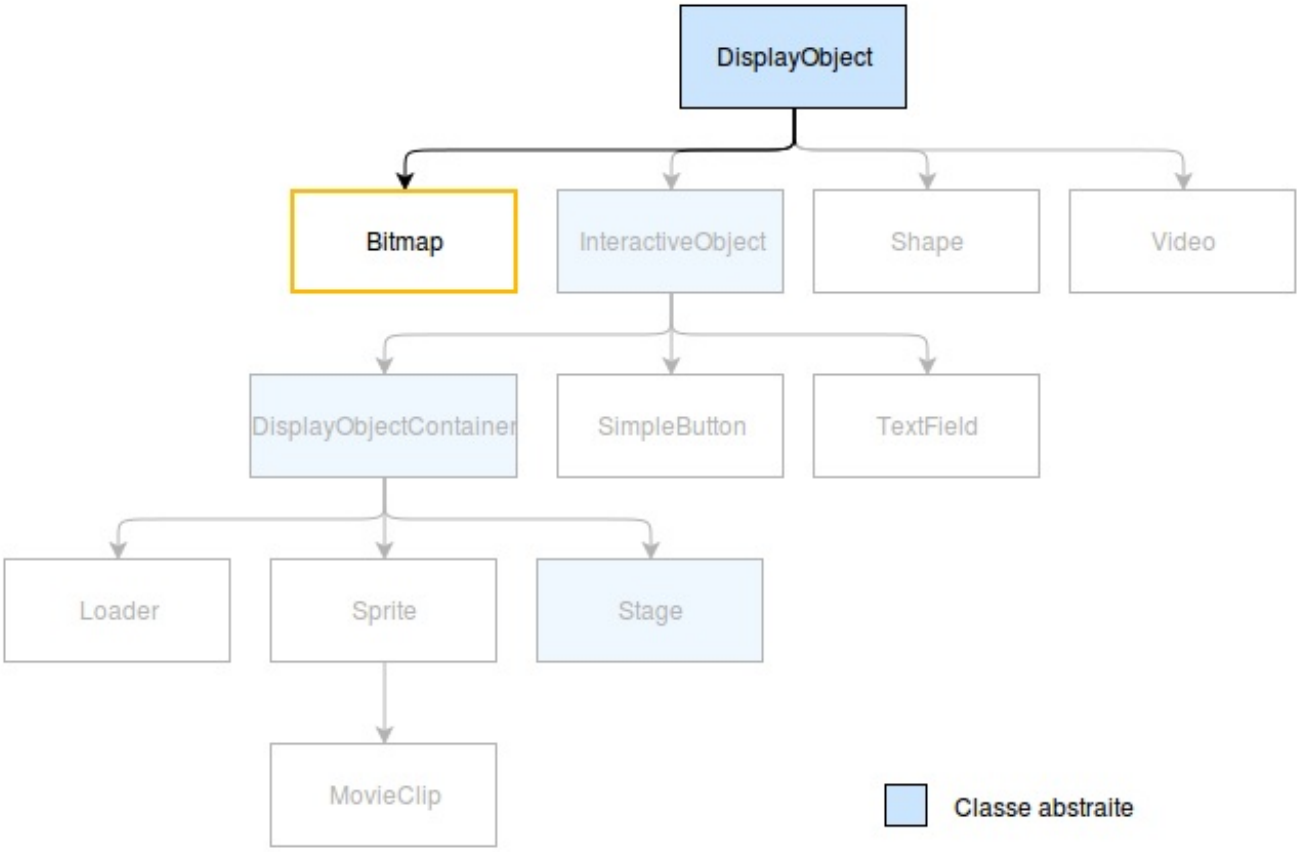
```
package
{
    /**
* Librairie contenant nos images embarquées.
* @author Guillaume CHAU
*/
    public class ImageLibrary
    {

        [Embed(source="../lib/img/zozor.png")]
        public static var Zozor:Class;
```

```
        }
    }
```

Notre librairie d'images embarquées est maintenant prête à l'emploi !

Afficher des images
La classe Bitmap

Pour afficher une image, nous allons utiliser la classe `Bitmap` (voir figure suivante).



La classe Bitmap dans l'arbre des classes d'affichage

Si nous utilisons maintenant le constructeur de la classe `Bitmap`, nous obtiendrons une image vide :

Code : Actionscript

```
// Image vide
var zozor:Bitmap = new Bitmap();
```

À la place, nous allons utiliser la classe contenu dans l'attribut statique que nous avons déclaré dans notre classe `ImageLibrary` :

Code : Actionscript

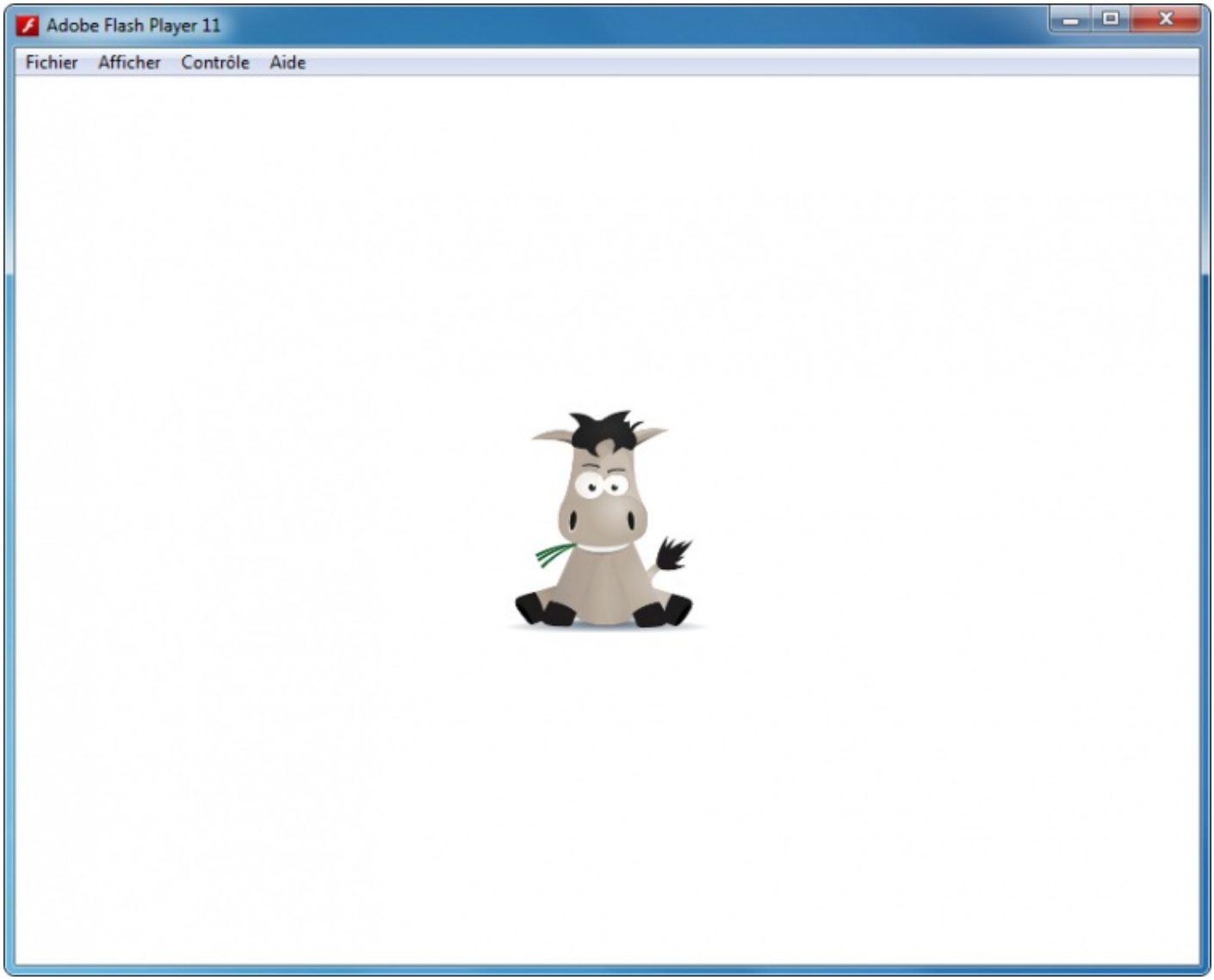
```
var zozor:Bitmap = new ImageLibrary.Zozor();
```

Voici le code complet pour afficher notre image de Zozor :

Code : Actionscript

```
// Création de l'image
var zozor:Bitmap = new ImageLibrary.Zozor();
addChild(zozor);

// Alignement au centre
zozor.x = (stage.stageWidth - zozor.width) / 2;
zozor.y = (stage.stageHeight - zozor.height) / 2;
```

Ce qui nous donne à l'écran la figure suivante.

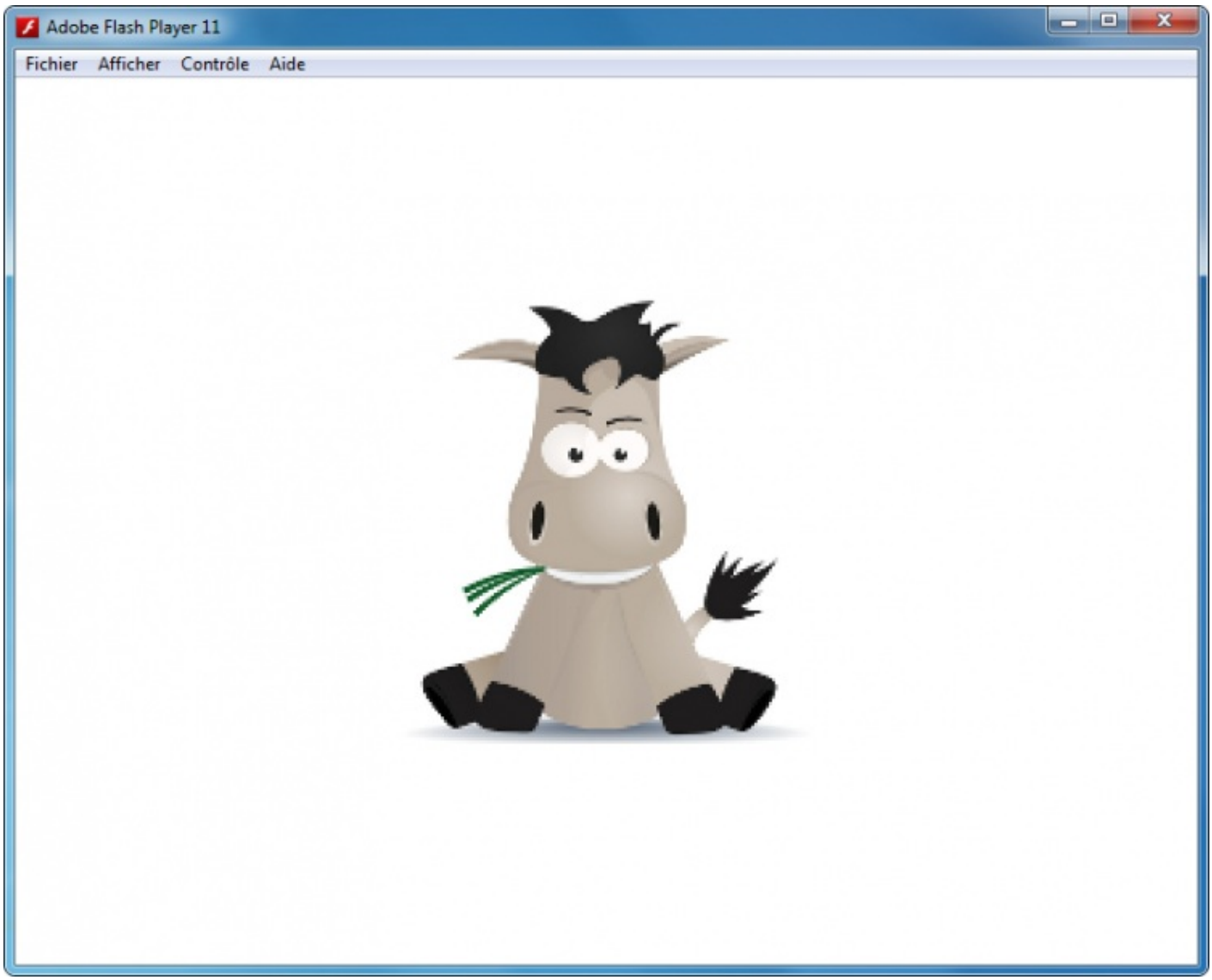


Zozor !

Redimensionnement

Je vous propose d'essayer de redimensionner notre image de Zozor : doublez sa taille sans regarder la solution, pour vérifier que vous vous souvenez bien des attributs nécessaires.

Vous devriez obtenir la figure suivante.



Zozor a pris du poids.

Voici une solution possible :

Code : Actionscript

```
// Création de l'image
var zozor:Bitmap = new ImageLibrary.Zozor();
addChild(zozor);

// Redimensionnement
zozor.scaleX = 2;
zozor.scaleY = zozor.scaleX;

// Alignement au centre
zozor.x = (stage.stageWidth - zozor.width) / 2;
zozor.y = (stage.stageHeight - zozor.height) / 2;
```

À l'aide des attributs `scaleX` et `scaleY` de la classe `DisplayObject` dont hérite la classe `Bitmap`, nous pouvons très facilement appliquer un coefficient multipliant la taille de notre objet d'affichage.

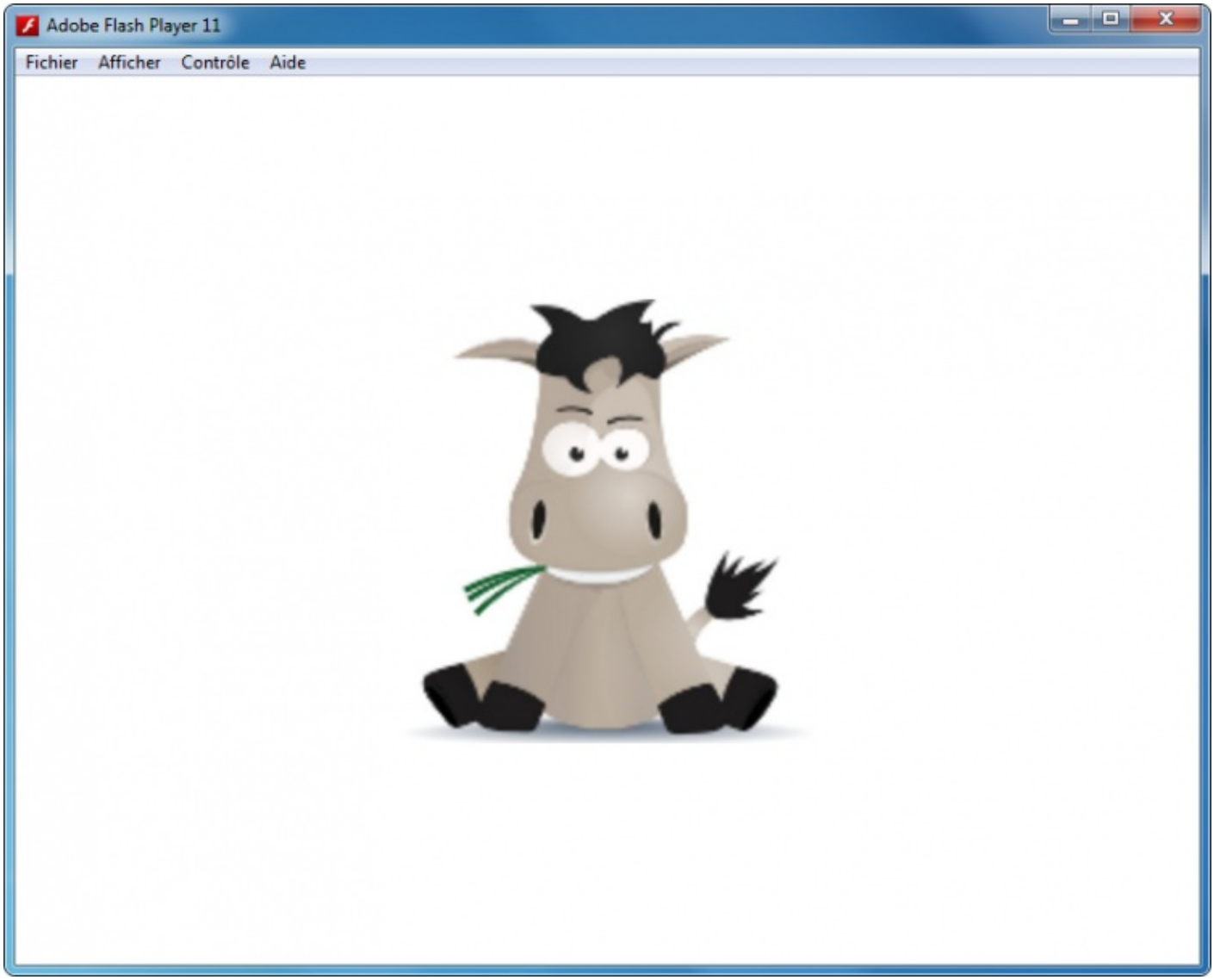
L'image est devenue affreuse avec ces gros pixels... Peux-on y remédier ?

Plus ou moins.
En vérité, il n'y a pas de formule magique pour rendre l'image aussi nette qu'elle était à sa taille d'origine, mais il est néanmoins possible d'améliorer le rendu obtenu lors de tels redimensionnements jusqu'à une certaine mesure. Pour cet effet, la classe `Bitmap` dispose d'un attribut `smoothing` qui indique si les pixels de l'image doivent être lissés si elle est redimensionnée :

Code : Actionscript

```
// Lissage des pixels
zozor.smoothing = true;
```

Ce qui nous donne à l'écran la figure suivante.



Les pixels sont lissés.

C'est déjà un peu mieux !
Mais l'effet de cet attribut se fait surtout ressentir lorsque l'on diminue la taille des images. Au lieu de doubler la taille de Zozo, mettons-la à 70%, sans utiliser le lissage :

Code : Actionscript

```
// Création de l'image
var zozor:Bitmap = new ImageLibrary.Zozor();
addChild(zozor);
```

```
// Lissage des pixels
//zozor.smoothing = true;

// Redimensionnement
zozor.scaleX = 0.7;
zozor.scaleY = zozor.scaleX;

// Alignement au centre
zozor.x = (stage.stageWidth - zozor.width) / 2;
zozor.y = (stage.stageHeight - zozor.height) / 2;
```

Ce qui nous donne la figure suivante.

Taille à 70% de l'original

Comme vous pouvez le voir, le rendu que nous obtenu est loin d'être agréable. Pour remédier à cela, réactivons le lissage :

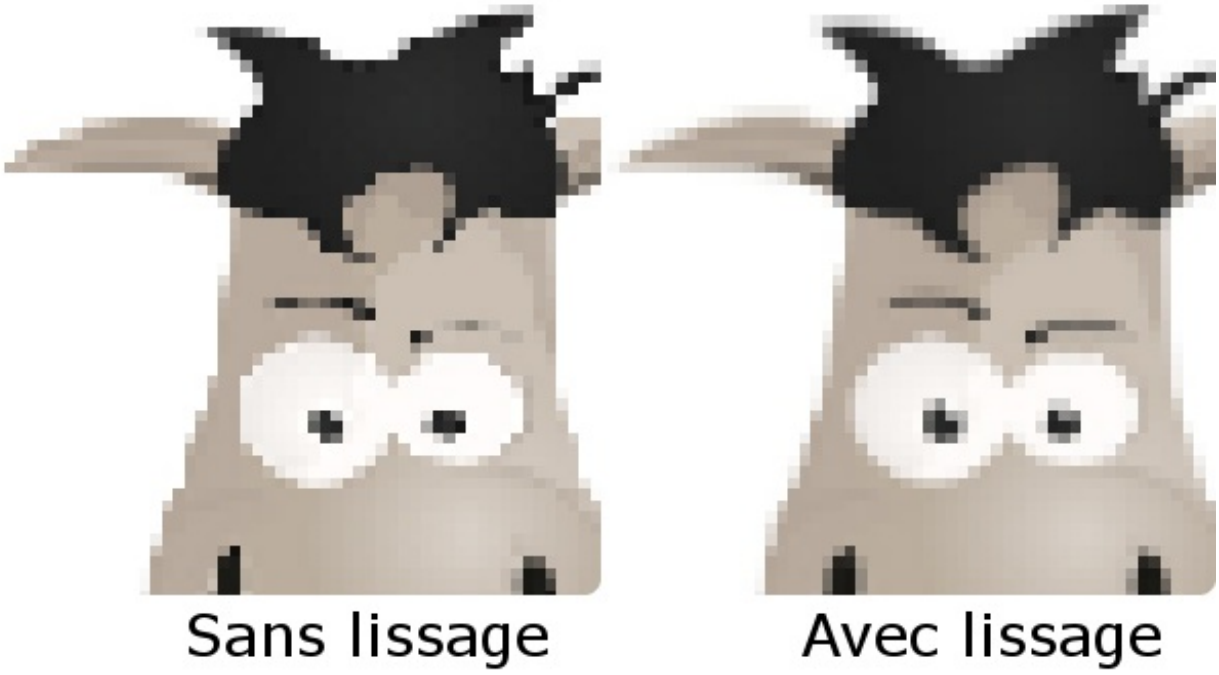
Code : Actionscript

```
// Création de l'image
var zozor:Bitmap = new ImageLibrary.Zozor();
addChild(zozor);

// Lissage des pixels
zozor.smoothing = true;

// Redimensionnement
zozor.scaleX = 0.7;
zozor.scaleY = zozor.scaleX;

// Alignement au centre
zozor.x = (stage.stageWidth - zozor.width) / 2;
zozor.y = (stage.stageHeight - zozor.height) / 2;
```

Ce qui nous donne à l'écran la figure suivante.

Taille à 70% avec lissage

C'est tout de même beaucoup mieux !
Pour mieux voir les différences entre les deux rendus, voici à la figure suivante une comparaison avec zoom.



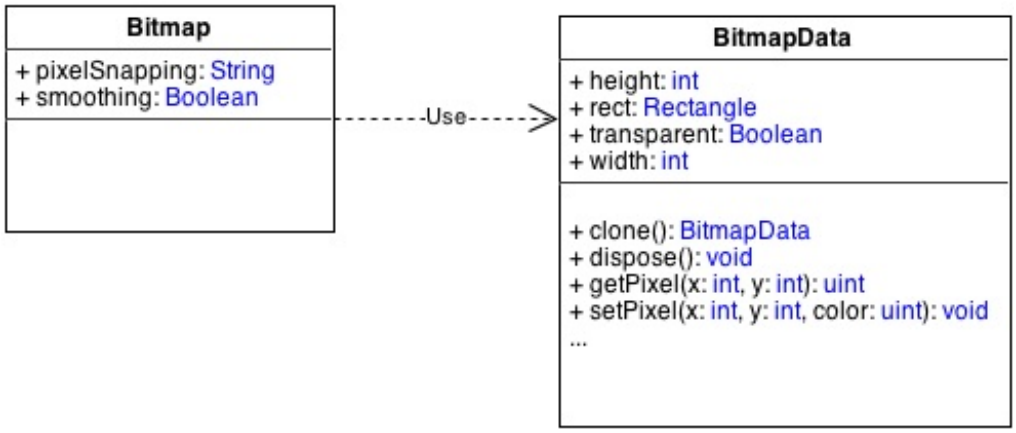
Comparaison avec et sans lissage

N'abusez pas du lissage si vous affichez beaucoup d'images : sinon, cela dégradera les performances de votre application.

Opérations sur les images

La classe `BitmapData`

Tout objet de la classe `Bitmap` dispose d'un attribut `bitmapData` de la classe `BitmapData` (voir figure suivante). Cet objet représente les pixels de l'image stockés en mémoire. L'instance de la classe `Bitmap` ne contient pas de pixels : elle n'est là que pour les afficher.



Tout objet de la classe Bitmap dispose d'un attribut bitmapData de la classe BitmapData

Il est possible d'utiliser un seul objet `BitmapData` pour plusieurs images `Bitmap`. N'hésitez pas à le faire si vous devez afficher plusieurs fois la même image, car cette astuce améliore grandement les performances d'affichage des images concernées.

Créer notre première image

Pour créer une image vide, nous pouvons créer une instance de la classe `Bitmap` sans lui passer de paramètre :

Code : Actionscript

```
// Création de l'image
var image:Bitmap = new Bitmap();
addChild(image);
```

Notre objet `image` n'a aucun pixel à sa disposition, donc il n'affiche rien et a une taille nulle :

Code : Actionscript

```
trace("taille de l'image : " + image.width + " x " + image.height);
```

Nous obtenons ceci dans la console :

Code : Console

```
taille de l'image : 0 x 0
```

Ne nous arrêtons pas là et créons notre première image grâce à la classe `BitmapData` :

Code : Actionscript

```
// Création des données de l'image (les pixels)
var imageData:BitmapData = new BitmapData(300, 300, false,
0x0000ff);
```

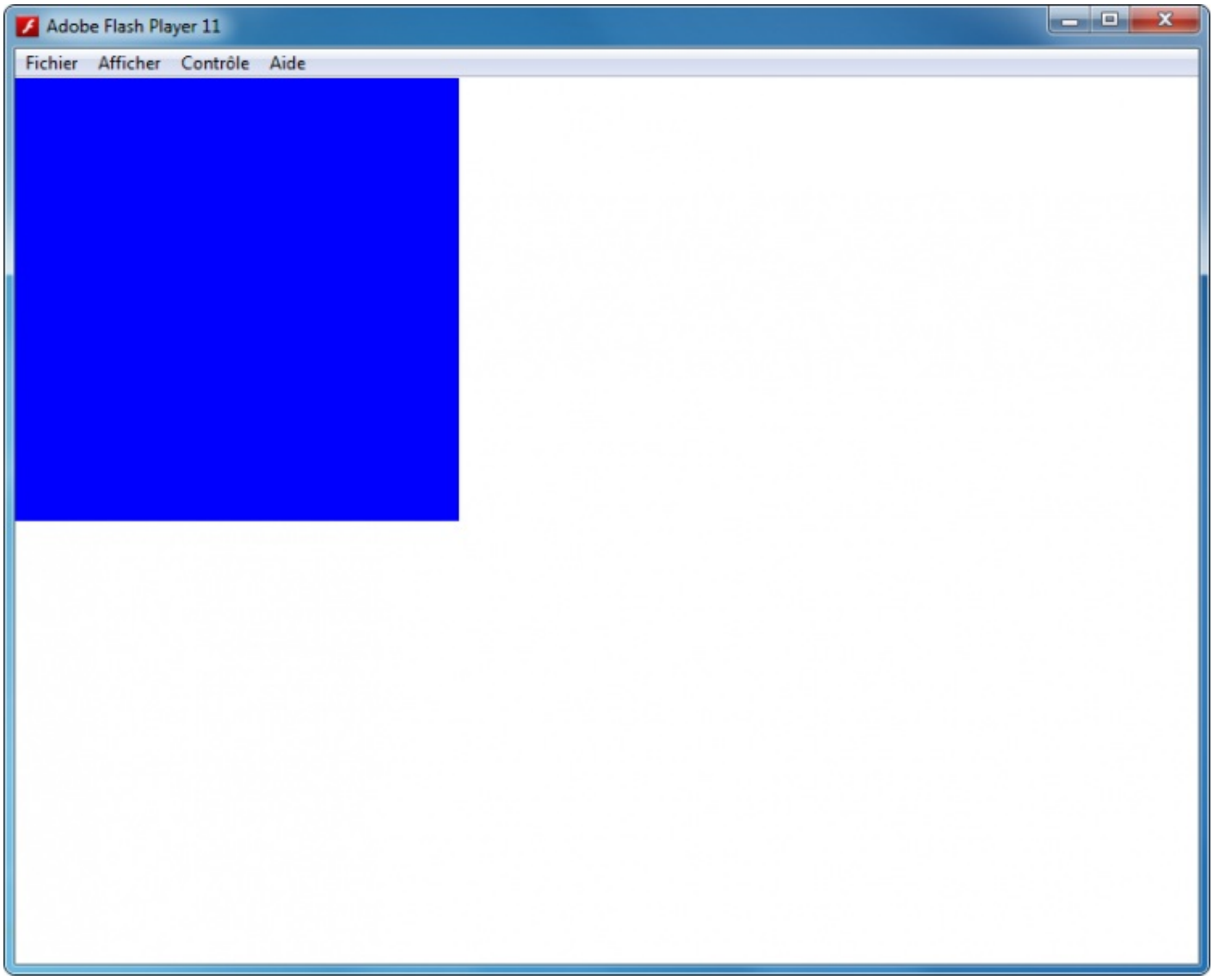
Le premier paramètre est la largeur du tableau de pixel (donc de l'image), puis le deuxième est sa hauteur. Ensuite, un booléen nous permet d'indiquer si l'image vide est transparente ou non ; enfin, nous passons la couleur de remplissage initiale.
Ici, notre image sera donc un carré de 300 pixels de côté, rempli de couleur bleue.

Ensuite, il faut dire à notre objet `image` d'utiliser les données contenues dans notre objet `imageData` :

Code : Actionscript

```
// Données pour l'image
image.bitmapData = imageData;
```

Nous avons désormais la figure suivante à l'écran.



Un carré bleu

Nous avons créé une image à partir de rien !

Dessiner sur des images

Nous allons maintenant nous atteler à rendre ce morne carré bleu un peu plus intéressant, en manipulant les pixels qui le composent.

Pipette

Nous avons à disposition d'une sorte de pipette, comme dans les logiciels de dessin, qui nous permet de connaître la couleur d'un pixel. Cette pipette est la méthode `getPixel(x, y)` de la classe `BitmapData`. En paramètre, nous lui fournissons les coordonnées du pixel dont nous voulons la couleur, et elle nous la renvoie (si nous donnons des coordonnées qui sortent de l'image, elle nous renvoie `0`).

Code : Actionscript

```
// Prenons la couleur d'un de nos pixels (celui du centre)
var couleur:uint = imageData.getPixel(150, 150);
trace("couleur du pixel : " + couleur.toString(16)); // Affiche: ff
(équivalent de 0000ff)
```

Il existe une deuxième pipette qui nous renvoie la couleur d'un pixel avec son opacité : `getPixel32(x, y)`.

Code : Actionscript

```
// Avec l'opacité
couleur = imageData.getPixel32(150, 150);
trace("couleur ARVB du pixel : " + couleur.toString(16)); //
Affiche: ff0000ff
```

Comme vous pouvez le remarquer, il y a deux caractères supplémentaires par rapport à une couleur classique : ce sont les deux premiers. Ils représentent l'opacité de la couleur en quelque sorte, notée en hexadécimal de `00` à `ff`, comme pour le rouge, le vert et le bleu. Ainsi, `0x00000000` est une couleur totalement transparente, `0x880000ff` représente du bleu à moitié transparent et `0xff0000ff` du bleu totalement opaque.

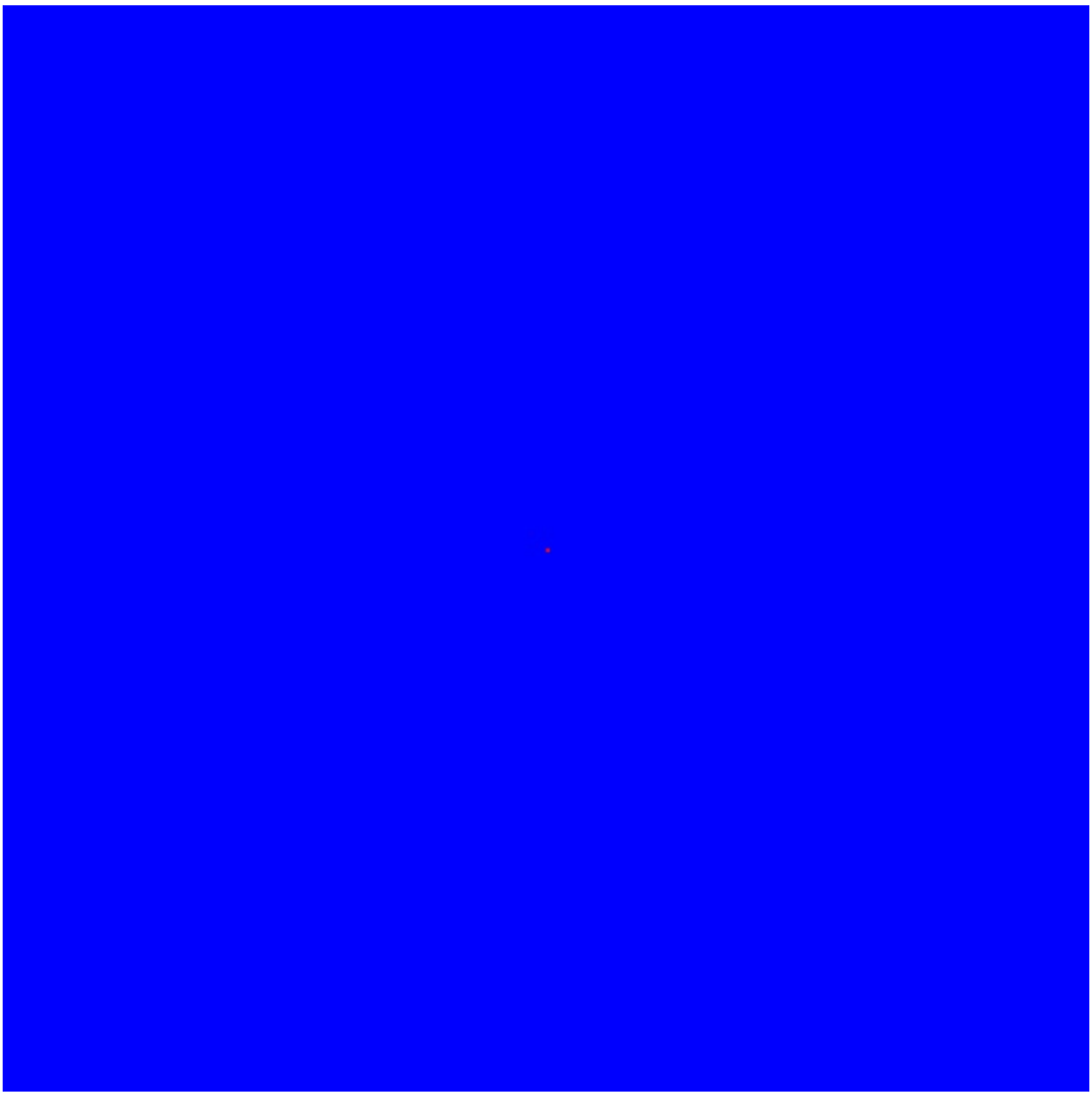
Colorier un pixel

Il est grand temps de barbouiller notre carré de couleur ! Commençons avec la méthode `setPixel(x, y, couleur)`, qui permet do colorier un pixel aux coordonnées que nous passons en paramètre, avec une certaine couleur (sans gestion de l'opacité) :

Code : Actionscript

```
// Colorions le pixel du centre en rouge
couleur = 0xff0000;
imageData.setPixel(150, 150, couleur);
```

Nous obtenons la figure suivante (j'ai agrandi l'image pour que l'on puisse mieux voir).



Pixel rouge au milieu de pixels bleus.

Comme tout à l'heure, il existe également la méthode `setPixel32(x, y, couleur)` qui permet de colorier un pixel avec une couleur contenant une valeur de transparence.

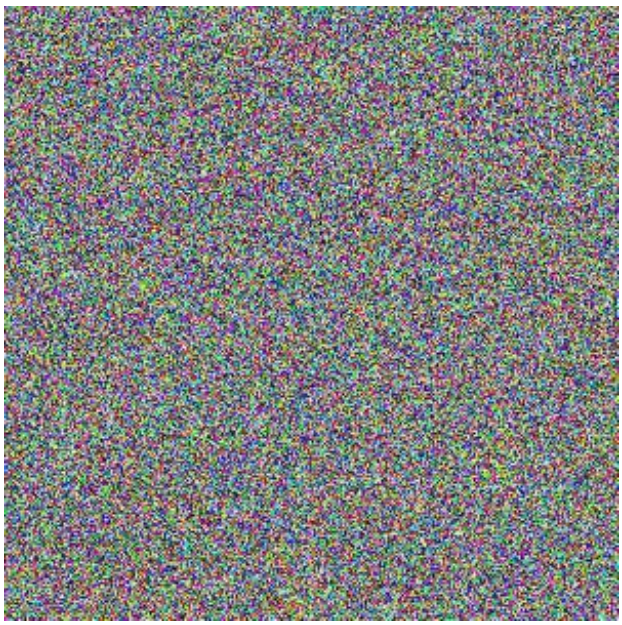
Exercice

Essayons-nous à plus de fantaisies ! Je vous propose un petit exercice : coloriez chaque pixel de notre carré avec une couleur aléatoire opaque. Essayez de ne pas regarder la solution trop vite ! 

Quelques conseils :

- utilisez deux boucles imbriquées pour "parcourir" les pixels de l'image,
- utilisez la méthode `Math.random()` pour générer un nombre aléatoire de `0` à `1` (que vous pouvez multiplier par `0xffffff` pour avoir une couleur aléatoire),
- la classe `BitmapData` dispose des attributs `width` et `height` permettant d'obtenir la taille de l'image stockée dans l'objet, un peu comme pour les objets d'affichage.

J'obtiens la figure suivante.



Un beau tas de pixels.

Ne regardez pas tout de suite la solution que je vous propose si vous n'avez pas réfléchi au code qu'il faut écrire. Essayez d'obtenir un bon résultat, et même si vous n'y arrivez pas, le fait d'avoir essayé est déjà un grand pas vers la maîtrise du langage.

Si vous avez réussi (ou que vous êtes sur le point de vous jeter par la fenêtre), vous pouvez comparer votre code avec le mien (je n'ai vraiment pas envie d'avoir votre mort sur la conscience) :

Code : Actionscript

```
// Création de l'image
var image:Bitmap = new Bitmap();
addChild(image);

// Création des données de l'image (les pixels)
var imageData:BitmapData = new BitmapData(300, 300, false,
0x0000ff);

// Données pour l'image
image.bitmapData = imageData;

///// Barbouillage !
// On parcourt chaque pixel de l'image
// Lignes
for (var x:int = 0; x < imageData.width; x ++)
{
 // Colonnes
 for (var y:int = 0; y < imageData.height; y ++)
 {
  // Couleur aléatoire
  couleur = Math.random() * 0xffffff;

  // Colorions le pixel actuel
  imageData.setPixel32(x, y, couleur);
 }
}
```

Et c'est tout ! Finalement, c'est plutôt simple non ?

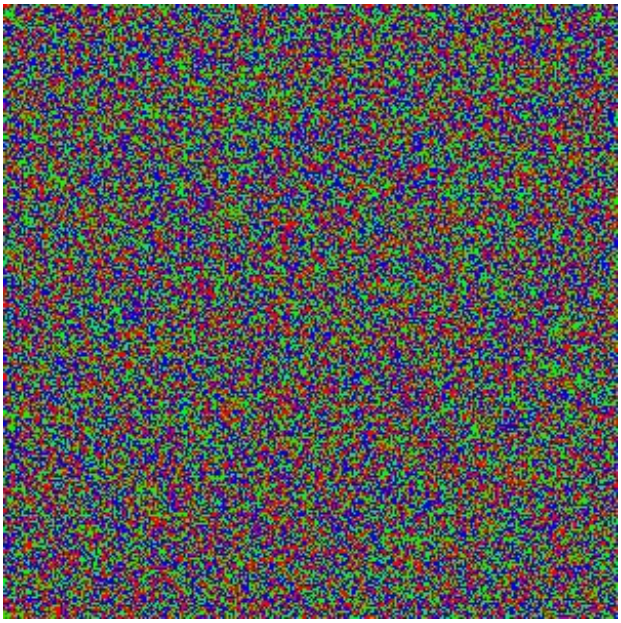
Exercice : variante

Et si nous corsions un peu les choses ? Modifions l'exercice : il faut désormais que tous les pixels de l'image soient remplis avec une couleur choisie aléatoirement parmi un choix limité : rouge, vert, bleu.

Quelques conseils :

- utilisez un tableau pour stocker les couleurs autorisées,
- réutilisez la méthode `Math.random()` et la propriété `length` de votre tableau pour choisir aléatoirement une couleur parmi celle du tableau.

J'obtiens cette fois-ci la figure suivante.



Pixels de trois couleurs

Solution :

Code : Actionscript

```
///// Barbouillage v 2.0
// On définit d'abord les couleurs possibles
var couleurs:Array = [0xff0000, 0x00ff00, 0x0000ff];

// On parcourt chaque pixel de l'image
// Lignes
for (var x:int = 0; x < imageData.width; x ++)
{
 // Colonnes
 for (var y:int = 0; y < imageData.height; y ++)
 {
  // Choix de la couleur
  var choix:int = Math.random() * couleurs.length;

  // Couleur choisie
  couleur = couleurs[choix];

  // Colorions le pixel actuel
  imageData.setPixel32(x, y, couleur);
 }
}
```

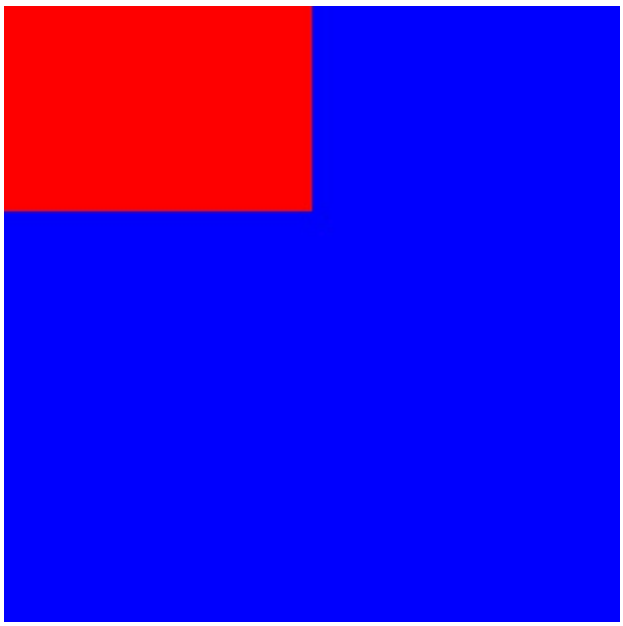
Remplissage un rectangle de couleur

Une autre méthode bien utile permet de remplir une zone rectangulaire avec une couleur (avec transparence), ce qui nous simplifie tout-de-même la vie. Il s'agit de `fillRect(rectangle, couleur)` de la classe `BitmapData`, qui prend en paramètre un objet de la classe `Rectangle` et une couleur avec transparence.

Code : Actionscript

```
// Traçons un rectangle rouge
imageData.fillRect(new Rectangle(0, 0, 150, 100), 0xffff0000);
```

Notre image ressemble désormais à la figure suivante.



Un rectangle rouge sur fond bleu

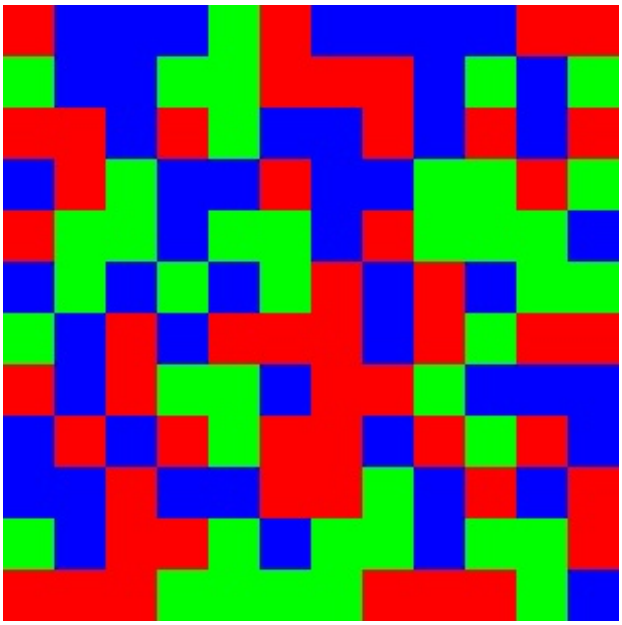
Exercice : deuxième variante

Hop ! Encore le même exercice ! Mais cette fois-ci, remplissons notre image avec de gros carrés de 25 pixels de côté, d'une couleur choisie aléatoirement parmi le même choix limité : rouge, vert, bleu.

Quelques conseils :

- utilisez un objet de la classe `Rectangle` pour stocker les coordonnées et la taille des carrés,
- et bien entendu, utilisez la méthode que nous venons de voir, `fillRect()` !

J'obtiens la figure suivante.



Le rendu

Et voici ce que j'ai écrit :

Code : Actionscript

```
///// Barbouillage v 3.0
// On définit d'abord les couleurs possibles
var couleur:uint;
var couleurs:Array = [0xff0000, 0x00ff00, 0x0000ff];

// On définit la taille des rectangles
var rectangle:Rectangle = new Rectangle(0, 0, 25, 25);

// On parcourt l'image pour la remplir de rectangles
// Lignes
for (rectangle.x = 0; rectangle.x < imageData.width; rectangle.x +=
rectangle.width)
{
    // Colonnes
    for (rectangle.y = 0; rectangle.y < imageData.height;
rectangle.y += rectangle.height)
    {
        // Choix de la couleur
        var choix:int = Math.random() * couleurs.length;

        // Couleur choisie
        couleur = couleurs[choix];

        // Colorions le rectangle actuel
        imageData.fillRect(rectangle, couleur);
    }
}
```

Comme vous pouvez le remarquer, j'ai utilisé l'objet `rectangle` directement dans les boucles **`for`** : le code en est d'autant plus logique et lisible.

En résumé

- Pour embarquer une image, il suffit d'utiliser l'instruction spéciale `Embed` et définir un attribut de type `Class`, comme pour les polices embarquées.
- L'affichage d'une image se fait via la classe `Bitmap`, dont la définition peut se faire grâce aux images embarquées.
- La classe `Bitmap` dispose d'une propriété `smoothing` qui permet de lisser les pixels d'une image, principalement utile après une transformation.
- L'ensemble des pixels d'une image sont stockés en mémoire à l'aide une instance de la classe `BitmapData`.
- La classe `Bitmap` possède donc une propriété nommée `bitmapData` de type `BitmapData`, dont l'édition est possible.

Filtres et modes de fusion

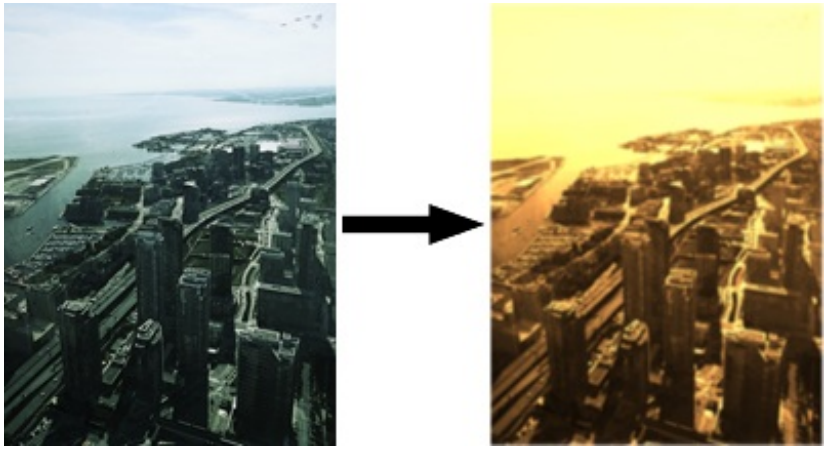
Dans ce chapitre, nous allons introduire les **filtres** et les **modes de fusion** ! Ils permettent de modifier facilement l'apparence de vos éléments graphiques à votre guise. Si certains en ont déjà entendu parler dans divers logiciels de graphisme, vous ne serez pas perdus puisque le principe est le même, et les effets sont assez semblables.

Enfin avant de vous lancer dans la lecture de ce chapitre, je dois vous mettre en garde. Effectivement, celui-ci est composé de deux glossaires qui regroupent donc l'ensemble des *filtres* et *modes de fusion*. Leur but est de pouvoir vous être utile plus tard lorsque vous en aurez besoin, ne passez donc pas trop de temps dessus pour le moment mais retenez simplement le principe général de ces effets.

Les filtres

Introduction

Une façon d'améliorer simplement le visuel de vos animation, est d'utiliser les différents **filtres** proposés par l'Actionscript. Il s'agit d'effets graphiques que vous pouvez ajouter à l'ensemble de vos éléments visuels. Vous disposez donc de *neuf* filtres que vous pouvez combiner selon vos préférences pour changer l'apparence d'un objet. Voici par exemple à la figure suivante une combinaison de deux filtres ajoutée à une image.



Deux filtres ont été ajoutés à une image

Vous constaterez donc qu'il est alors possible de radicalement transformer le rendu final d'une scène. Parmi les filtres, vous trouverez des effets :

- de flou
- d'ombre portée
- de lueur
- de biseau
- et bien d'autres.

Avant de vous énumérer l'ensemble des filtres disponibles je vous propose de voir ensemble le principe d'utilisation de ceux-ci.

Création d'un exemple

Pour comprendre le fonctionnement des filtres, nous allons prendre un exemple. Je suggère donc que nous reprenions l'image ainsi que l'effet précédent. La figure suivante représente l'image originale, telle qu'elle est chargée à l'intérieur de l'animation. Vous pouvez évidemment travailler sur une image différente.

L'image originale

Nous allons donc charger notre image comme vous savez maintenant le faire :

Code : Actionscript

```
[Embed(source = 'images/Photo.png')]
private var Photo:Class;
```

Puis nous allons l'instancier et l'ajouter à la liste d'affichage :

Code : Actionscript

```
var maPhoto:Bitmap = new Photo();
this.addChild(maPhoto);
```

Ajout de filtres

Chacun des neufs filtres est en fait défini par sa propre classe. Ainsi ils possèdent chacun leurs propres propriétés permettant de paramétrer l'effet correspondant. Dans l'exemple précédent, nous avons utilisé les deux classes `BlurFilter` et `ColorMatrixFilter`.
Nous apprendrons à utiliser ces classes plus tard, c'est pourquoi je vous propose directement le code de ces deux effets.

Flou

Code : Actionscript

```
var flou:BlurFilter = new BlurFilter();
flou.blurX = 2;
flou.blurY = 2;
```

```
flou.quality = BitmapFilterQuality.HIGH;
```

Correction colorimétrique

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(0.75, 0.75, 0.75, 0, 0); // Rouge
matrix.push(0.50, 0.50, 0.50, 0, 0); // Vert
matrix.push(0.25, 0.25, 0.25, 0, 0); // Bleu
matrix.push(0, 0, 0, 1, 0); // Alpha
var correctionCouleur:ColorMatrixFilter = new
ColorMatrixFilter(matrix);
```

Application des filtres

Tout objet `DisplayObject` possède une propriété `filters` de type `Array`. Il est ainsi possible d'ajouter des filtres à sa guise en manipulant ce tableau.
Ajoutons donc nos deux filtres à l'initialisation de notre tableau `filters` :

Code : Actionscript

```
maPhoto.filters = new Array(correctionCouleur, flou);
```

Nous obtenons ainsi notre image avec une correction de couleur et un flou, comme le montre la figure suivante.

L'image a été modifiée

Glossaire des filtres
Les filtres de base

Flou

Pour flouter un élément graphique, nous disposons de la classe `BlurFilter` :

Code : Actionscript

```
var flou:BlurFilter = new BlurFilter();
flou.blurX = 5;
flou.blurY = 5;
flou.quality = BitmapFilterQuality.MEDIUM;
```

Le flou est ainsi définit par les propriétés suivantes :

- `blurX` : quantité de flou horizontal
- `blurY` : quantité de flou vertical
- `quality` : qualité du flou définie par les valeurs de la classe `BitmapFilterQuality`.

Le résultat se trouve à la figure suivante.

L'image est floutée

Ombre portée

Les ombres portées sont définies grâce à la classe `DropShadowFilter` :

Code : Actionscript

```
var ombre:DropShadowFilter = new DropShadowFilter();
ombre.color = 0x333333;
ombre.distance = 5;
ombre.angle = -30;
```

Pour paramétrer votre ombre portée, utilisez les propriétés suivantes :

- `color` : couleur de l'ombre
- `distance` : distance « fictive » de l'objet à la surface derrière elle
- `angle` : orientation de la lumière et donc de l'ombre.

Le résultat se trouve à la figure suivante.

Une ombre portée a été ajoutée

Lueur simple

Il est possible de créer une lueur ou un rayonnement derrière l'objet grâce à la classe `GlowFilter` :

Code : Actionscript

```
var lueur:GlowFilter = new GlowFilter();
lueur.color = 0xFFAA66;
lueur.blurX = 15;
lueur.blurY = 15;
lueur.quality = BitmapFilterQuality.MEDIUM;
```

Nous retrouvons ici des propriétés semblables à celles que nous avons déjà vu précédemment, auxqelles vous pouvez rajouter différentes propriétés :

- `alpha` : opacité de la lueur
- `strength` : intensité du rayonnement
- `inner` : indique si l'ombre est interne ou non
- `hideObject` : indique si l'objet doit être visible ou non.

Le résultat se trouve à la figure suivante.

Une lueur simple

Lueur en dégradé

L'effet de rayonnement vu juste avant peut être personnalisé en ajoutant un dégradé à la lueur. Pour cela, nous disposons de la classe `GradientGlowFilter` :

Code : Actionscript

```
var lueur:GradientGlowFilter = new GradientGlowFilter();
lueur.distance = 0;
lueur.angle = 45;
lueur.colors = [0xFF0000, 0x333333];
lueur.alphas = [0, 1];
lueur.ratios = [0, 255];
lueur.blurX = 15;
lueur.blurY = 15;
lueur.strength = 2;
lueur.quality = BitmapFilterQuality.HIGH;
lueur.type = BitmapFilterType.OUTER;
```

Les nouveautés sont ici la définition du dégradé grâce aux propriétés `colors`, `alphas` et `ratios`, ainsi que la définition de la lueur en interne ou en externe par le biais de la classe `BitmapFilterType`.

Le résultat se trouve à la figure suivante.

Une lueur en dégradé

Biseau simple

Pour donner du relief à un objet, vous pouvez définir un biseau à l'aide de la classe `BevelFilter` :

Code : Actionscript

```
var biseau:BevelFilter = new BevelFilter();
biseau.distance = 5;
biseau.angle = 45;
biseau.highlightColor = 0xFFFFFF;
biseau.highlightAlpha = 0.5;
biseau.shadowColor = 0x000000;
biseau.shadowAlpha = 0.5;
biseau.blurX = 3;
biseau.blurY = 3;
biseau.strength = 3;
biseau.quality = BitmapFilterQuality.HIGH;
biseau.type = BitmapFilterType.INNER;
```

Pour définir ce filtre, vous aurez principalement besoin de définir la distance de l'effet ainsi que des effets de lueurs et d'ombres définis par les couleurs et opacités `highlightColor`, `highlightAlpha`, `shadowColor` et `shadowAlpha`.

Le résultat se trouve à la figure suivante.

Un biseau simple

Biseau en dégradé

Une variante de biseau existe grâce à l'utilisation d'un dégradé dans la classe `GradientBevelFilter` :

Code : Actionscript

```
var biseau:GradientBevelFilter = new GradientBevelFilter();
biseau.distance = 6;
biseau.angle = 60;
biseau.colors = [0x000000, 0x006666, 0xFFFFFF];
biseau.alphas = [1, 0, 1];
biseau.ratios = [0, 128, 255];
biseau.blurX = 6;
biseau.blurY = 6;
biseau.quality = BitmapFilterQuality.HIGH;
```

Là encore, la principale différence par rapport à la version simple est la définition d'un dégradé de couleurs.

Le résultat se trouve à la figure suivante.

Un biseau en dégradé

Correction de couleurs

Un peu de théorie

Pour effectuer des corrections au niveau des couleurs, il existe la classe `ColorMatrixFilter`. Comme son nom l'indique, elle est composée d'une matrice qui permet de redéfinir la couleur d'un pixel. Ainsi les lignes redéfinissent, dans l'ordre, les composantes suivantes d'un pixel : rouge, vert, bleu et alpha. Les différentes composantes de celles-ci peuvent alors s'écrire

suivant la forme suivante :

$$f_{canal}(x,y) = \alpha_R \times s_{rouge}(x,y) + \alpha_V \times s_{vert}(x,y) + \alpha_B \times s_{bleu}(x,y) + \alpha_A \times s_{alpha}(x,y) + \alpha$$

Où :

- $s_{rouge}(x,y)$, $s_{vert}(x,y)$, $s_{bleu}(x,y)$ et $s_{alpha}(x,y)$ sont les composantes rouge, vert, bleu et alpha du pixel source
- α_R, α_V, α_B et α_A représentent leur contribution dans la valeur du canal de sortie
- α est une constante.

En réalité, cette matrice est décrite à l'intérieur d'un tableau de 20 éléments. Chaque ligne est alors définie par une succession de 5 coefficients. Voici donc notre « pseudo-matrice » A, où les colonnes correspondent aux composantes du pixel d'origine et les lignes aux composantes du pixel final :

	s_{rouge}	s_{vert}	s_{bleu}	s_{alpha}	
f_{rouge}	A[0]	A[1]	A[2]	A[3]	A[4]
f_{vert}	A[5]	A[6]	A[7]	A[8]	A[9]
f_{bleu}	A[10]	A[11]	A[12]	A[13]	A[14]
f_{alpha}	A[15]	A[16]	A[17]	A[18]	A[19]

En suivant cette logique, l'image serait inchangée par application de la matrice suivante :

$$\begin{matrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{matrix}$$

Afin de mieux comprendre, je vous propose à la suite divers exemples d'utilisation de cette classe `ColorMatrixFilter`.

Saturation

Cette technique consiste à *saturer* les couleurs afin d'obtenir une image finale en noir et blanc. Il faut alors que chaque pixel dispose au final de la même quantité de rouge, de vert et de bleu. Voici comment réaliser cela :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(0.33, 0.33, 0.33, 0, 0); // Rouge
matrix.push(0.33, 0.33, 0.33, 0, 0); // Vert
matrix.push(0.33, 0.33, 0.33, 0, 0); // Bleu
matrix.push(0, 0, 0, 1, 0); // Alpha
var correctionCouleur:ColorMatrixFilter = new
ColorMatrixFilter(matrix);
```

Le résultat se trouve à la figure suivante.


Un effet de saturation

Pour ne pas modifier la luminosité globale de l'image, veillez à avoir un total de 1 sur l'ensemble d'une ligne. Autrement les valeurs originalement comprise entre 0 et 255 seront disproportionnées et l'image n'aura pas le même rendu au final.

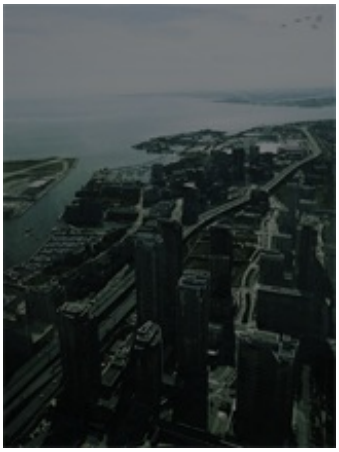
Luminosité

En parlant de luminosité, il est justement possible de la contrôler en jouant sur la valeur ces coefficients. Nous pouvons alors diminuer cette luminosité par exemple :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(0.5, 0, 0, 0, 0); // Rouge
matrix.push(0, 0.5, 0, 0, 0); // Vert
matrix.push(0, 0, 0.5, 0, 0); // Bleu
matrix.push(0, 0, 0, 1, 0); // Alpha
var correctionCouleur:ColorMatrixFilter = new
ColorMatrixFilter(matrix);
```

Le résultat se trouve à la figure suivante.


Il est possible de jouer sur la luminosité

Le réglage de la luminosité est le seul cas où vous pouvez modifier la somme totale des coefficients. Pour tout le reste, respectez au maximum cette règle.

Teinte

La teinte est plus difficile à gérer, car la « rotation » des couleurs est généralement suivie d'une saturation.
Toutefois je vous donne l'exemple d'un décalage de 120° dans les couleurs, ce qui revient à décaler le vert au rouge, le bleu au

vert, etc :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(0, 0, 1, 0, 0); // Rouge
matrix.push(1, 0, 0, 0, 0); // Vert
matrix.push(0, 1, 0, 0, 0); // Bleu
matrix.push(0, 0, 0, 1, 0); // Alpha
var correctionCouleur:ColorMatrixFilter = new
ColorMatrixFilter(matrix);
```

Le résultat se trouve à la figure suivante.

Il est possible de « décaler » les couleurs

Pour ce genre de manipulations, je vous conseille plutôt de passer par votre logiciel de graphisme préféré.

Négatif

Un effet négatif consiste à inverser les couleurs, c'est-à-dire soustraire les valeurs d'origine à la valeur maximale 255.
Dans ce cas, nous aurons alors besoin d'utiliser une constante, comme vous pouvez le voir ci-dessous :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(-1, 0, 0, 0, 255); // Rouge
matrix.push(0, -1, 0, 0, 255); // Vert
matrix.push(0, 0, -1, 0, 255); // Bleu
matrix.push(0, 0, 0, 1, 0); // Alpha
var correctionCouleur:ColorMatrixFilter = new
ColorMatrixFilter(matrix);
```

Le résultat se trouve à la figure suivante.

L'image en négatif

Les constantes ne sont pas des coefficients comme les précédents. Ainsi ceux-ci ne s'utilisent pas entre -1 et 1 comme les autres, mais plutôt entre -255 et 255.

Convolution

Introduction

La convolution utilisée par la classe `ConvolutionFilter` fonctionne également à l'aide d'une matrice. Néanmoins, celle-ci est une représentation d'un pixel et de ses voisins proches. Cette matrice peut théoriquement être de dimension quelconque, toutefois nous nous contenterons d'une matrice 3×3 pour la suite.
Ainsi la valeur centrale représente le pixel actuel et les autres valeurs les pixels alentours. La couleur finale d'un pixel est alors déterminée à partir de sa valeur initiale et de celles des pixels voisins.
Une image intacte est donc composé uniquement de la valeur `1` au centre de la matrice :

$$\begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix}$$

Ici, la valeur de chaque pixel est une combinaison de la valeurs de plusieurs pixels. Ainsi il est également important d'avoir un total égale à 1 pour conserver la luminosité globale de l'image initiale.

Avant de vous montrer différents exemples d'application, je vous présente ici la manière d'utiliser ce filtre en Actionscript :

Code : Actionscript

```
var matrix:Array = new Array();
// Définition de la matrice
var convolution:ConvolutionFilter = new ConvolutionFilter();
convolution.matrixX = 3;
convolution.matrixY = 3;
convolution.matrix = matrix;
```

Vous remarquerez alors que la matrice est également définie à l'aide d'un tableau de type `Array`, dont il est nécessaire de spécifier les dimensions à l'aide des propriétés `matrixX` et `matrixY`.

Flou

L'application la plus simple est donc de générer un flou en « moyennant » la valeur du pixel central à celle des pixels voisins :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(0, 0.2, 0);
matrix.push(0.2, 0.2, 0.2),
matrix.push(0, 0.2, 0);
```

Le résultat se trouve à la figure suivante.

L'image est floue

Il pourrait également être possible de prendre en compte la valeur des pixels en diagonale pour la génération de ce flou.

Détection de contours

La convolution est principalement utile pour détecter les contours à l'intérieur d'une image. Le principe est alors de retrancher la valeur des pixels voisins, et ainsi révéler les zones où les pixels proches n'ont pas une valeur identiques. Voici donc comment détecter des contours :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(0, -1, 0);
matrix.push(-1, 4, -1),
matrix.push(0, -1, 0);
```

Le résultat se trouve à la figure suivante.

Détection des contours

Notez que cette opération peut également être effectuée uniquement en horizontal ou en vertical, on même en diagonal !

Accentuation de contours

Si vous avez bien regardé la matrice de détection de contours, vous noterez que la somme des valeurs est nulle. Nous avons alors perdu des informations concernant l'image. Ainsi en rajoutant 1 au centre, il est possible de transformer cette détection de contours en accentuation de contours. Voyez donc ce que cette nouvelle matrice provoque :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(0, -1, 0);
matrix.push(-1, 5, -1),
matrix.push(0, -1, 0);
```

Le résultat se trouve à la figure suivante.

Accentuation des contours

Estampage

Cette détection de contours permet également de créer un effet d'estampage, pour faire ressortir le « relief » de l'image. Voici comment procéder :

Code : Actionscript

```
var matrix:Array = new Array();
matrix.push(-2, -1, 0);
matrix.push(-1, 1, 1),
matrix.push(0, 1, 2);
```

Le résultat se trouve à la figure suivante.


Estampage de l'image

Dans l'exemple précédent nous avons des valeurs négatives dans le coin supérieur gauche et des valeurs positives dans le coin opposé, ce qui a pour effet de « creuser » l'image. En inversant ces valeurs, vous obtiendrez donc l'effet inverse.

Mappage de déplacement

Enfin, la classe `DisplacementMapFilter` est le dernier filtre en Flash. Celui-ci permet d'utiliser ce qu'on appelle une carte de déplacement pour déformer une image.
Le déplacement est alors créé à partir des différents canaux de cette « carte ». Dans l'exemple que nous verrons par la suite, nous avons ainsi défini le canal de couleur bleu pour l'affectation de la position en x, et le canal de couleur rouge de celui en y.
Voici donc notre image de mappage du déplacement à la figure suivante.

<image legende="" legendevisible="oui">http://uploads.siteduzero.com/files/407001_408000/407091.png</image>

Vous noterez que le déplacement se fait suivant la valeur du canal de couleur correspondant. C'est pourquoi la valeur 127 n'affecte aucun déplacement, contrairement aux valeurs 255 et 0 qui provoquent un déplacement maximal dans des directions opposées.

Voici donc comment utiliser cette image de mappage du déplacement chargée grâce à la classe `Class` sous le nom de `Deplacement` :

Code : Actionscript

```
var monDeplacement:Bitmap = new Deplacement();
// Définition de la carte de déplacement
var map:BitmapData = new BitmapData(monDeplacement.width,
monDeplacement.height);
map.draw(monDeplacement);
// Création du filtre
var deplacement:DisplacementMapFilter = new DisplacementMapFilter();
deplacement.mapBitmap = map;
deplacement.componentX = BitmapDataChannel.BLUE;
deplacement.componentY = BitmapDataChannel.RED;
deplacement.scaleX = 60;
deplacement.scaleY = 60;
deplacement.mode = DisplacementMapFilterMode.IGNORE;
```

Comme vous avez certainement deviné, les canaux de couleurs sont choisis grâce à la classe `BitmapDataChannel` et l'échelle d'affectation est spécifiée à l'aide des propriétés `scaleX` et `scaleY`. Enfin les constantes de la classe `DisplacementMapFilterMode` permettent de gérer l'affectation des pixels sur les bords de l'image.

Le résultat se trouve à la figure suivante.

<image legende="" legendevisible="oui">http://uploads.siteduzero.com/files/407001_408000/407092.png</image>

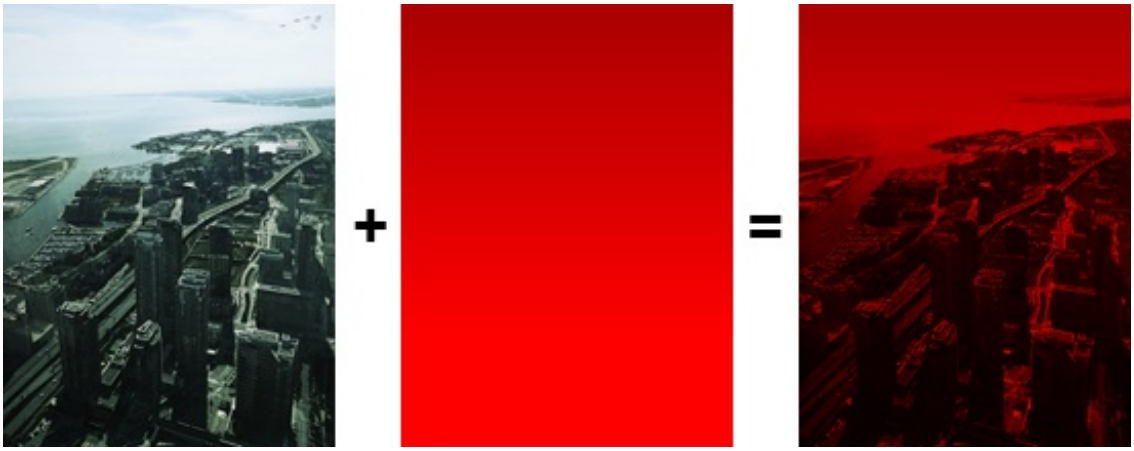
Les modes de fusion

Définition

Un autre moyen d'améliorer rapidement et facilement le rendu final de vos objets visuels est l'utilisation de **modes de fusion** ! Lorsqu'on parle de fusion, il s'agit en fait de « mixer » différents éléments graphiques *ensemble*. Le principe est alors de gérer la manière dont ces différents objets seront fusionnés dans le rendu final. Les modes de fusion sont donc également une manière rapide de transformer l'apparence de vos objets d'affichage.

Fusion de calque

La première utilité de ces modes de fusion est de pouvoir combiner différents objets superposés, nommés communément **calques**, en fusionnant leurs couleurs.
Voici à la figure suivante un exemple de fusion entre deux calques.

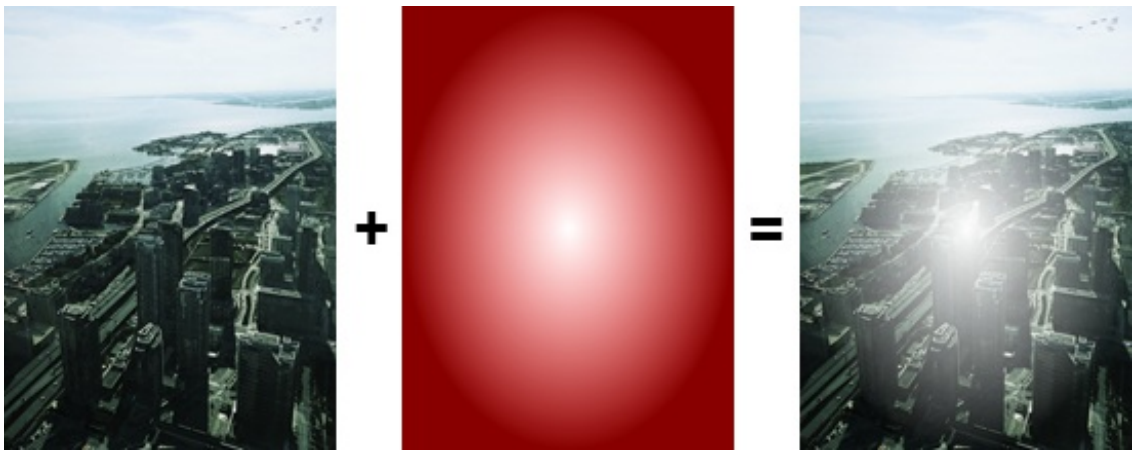


Fusion de deux images

Pour bien distinguer les *filtres* et les *modes de fusions*, retenez qu'un filtre s'applique à un *unique* objet, contrairement aux modes de fusions qui nécessitent au moins deux objets pour pouvoir être actifs.

Fusion de transparence

La seconde utilisation de ces modes de fusion est la fusion de transparence. Le principe ici est alors d'utiliser les niveaux de transparence d'un calque pour masquer certaines parties d'un second élément.
Voici un exemple :



Fusion de transparence

Une autre manière de gérer l'opacité de certaines zones d'un élément est l'utilisation de **masques**, comme nous le verrons dans le chapitre suivant.

Mise en place

La fusion de calques peut être réalisée entre n'importe quels objets héritant de la classe `DisplayObject`. Ceux-ci disposent alors d'une propriété nommée `blendMode` pour définir le mode de fusion désiré. Cette définition est alors réalisée à l'aide des constantes de la classe `BlendMode`.

La définition d'un mode de fusion se fait en réalité uniquement sur un seul calque. Vous devez alors l'appliquer sur l'objet possédant l'index d'affichage le plus grand, autrement l'effet de sera pas visible.

Voici donc comment ajouter le mode de fusion nommée `DARKEN` à l'objet `monCalque` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.DARKEN;
```

Toutefois en ce qui concerne les fusions de transparence, vous aurez également besoin d'imposer la création d'un groupe de transparences pour l'objet d'affichage. Cela consiste en fait à appliquer un mode de fusion de type `LAYER` au conteneur :

Code : Actionscript

```
this.blendMode = BlendMode.LAYER;
```

Voilà, vous connaissez tout sur les modes de fusion à présent. Il ne nous reste plus qu'à faire un tour des *treize* modes de fusion disponibles.
Je vous propose donc un petit glossaire pour la suite.

Glossaire de modes de fusion

Le mode de fusion par défaut

En réalité tout objet d'affichage possèdent obligatoirement un *seul et unique* mode de fusion. Ainsi il existe un mode de fusion nommé `NORMAL` qui est appliqué par défaut à tout objet. Voici donc comment redéfinir le mode de fusion :

Code : Actionscript

```
monCalque.blendMode = BlendMode.NORMAL;
```

Le résultat se trouve à la figure suivante.

Fusion par défaut

Ce mode n'a aucune utilité, si ce n'est de revenir au mode de fusion par défaut.

Les fusions de calques

Addition

Comme son nom l'indique, ce mode de fusion va additionner les différents canaux de couleurs du calque à ceux de l'arrière-plan. Pour utiliser celui-ci, vous disposez le la constante `ADD` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.ADD;
```

Le résultat se trouve à la figure suivante.

Fusion par addition

Ce mode de fusion aura donc tendance à éclaircir l'arrière-plan. Celui-ci peut être utilisé pour créer des transitions entre deux objets en utilisant un fondu d'éclaircissement.

Obscurcir

Ce mode de fusion permet de sélectionner les valeurs les plus faibles des canaux de couleurs entre le calque et l'arrière-plan. Il est défini par la constante `DARKEN` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.DARKEN;
```

Le résultat se trouve à la figure suivante.

L'image est obscurcie

Bien évidement, en l'utilisant vous allez obscurcir l'arrière plan. L'assombrissement est toutefois moins intense qu'avec le mode de fusion `MULTIPLY`.

Différence

Le principe ici est de soustraire pour chacun des canaux de couleurs la valeur la plus sombre des deux de la plus claire. Voici la constante concernée :

Code : Actionscript

```
monCalque.blendMode = BlendMode.DIFFERENCE;
```

Le résultat se trouve à la figure suivante.

Fusion par différence de couleur

L'utilisation de ce mode de fusion n'est pas très courante, car il est difficile de prévoir l'apparence finale. Utilisez donc celui-ci avec précaution !

Lumière crue

Le principe de ce mode de fusion est relativement complexe. Néanmoins vous pouvez retenir qu'il permet de faire ressortir et d'assombrir les ombrages d'une image. Le nom de la constante qui le définit est `HARDLIGHT` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.HARDLIGHT;
```

Le résultat se trouve à la figure suivante.

Les ombres sont mises en valeur

Notez que celui-ci a tendance a masquer les couleurs claires de l'arrière-plan.

Négatif

Le principe de cet effet, que nous avons déjà vu, est d'inverser les valeurs sur chacun des canaux de couleurs. Il est défini par la constante `INVERT` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.INVERT;
```

Le résultat se trouve à la figure suivante.

Inversion de la couleur de chaque pixel

La couleur d'origine du calque, où le mode de fusion est appliqué, n'a ici aucun effet sur le rendu final. Il s'agit également d'un mode de fusion peu courant.

Éclaircir

Dans ce mode de fusion, les valeurs de canaux de couleurs de couleurs retenues sont celles qui sont les plus claires entre le calque et l'arrière-plan. Il s'agit du mode de fusion opposé à `DARKEN`. Voici sa constante :

Code : Actionscript

```
monCalque.blendMode = BlendMode.LIGHTEN;
```

Le résultat se trouve à la figure suivante.

Éclaicissement de l'image

Également, celui-ci est moins éclaircissant que le mode de fusion `ADD`.

Produit

Les valeurs des canaux de couleurs du calque et de l'arrière-plan sont multipliées entre elles, puis sont « normalisées ». La constante correspondante est `MULTIPLY` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.MULTIPLY;
```

Le résultat se trouve à la figure suivante.

Multiplication des couleurs

L'effet produit est un assombrissement de l'arrière-plan. Ce mode de fusion proche de `DARKEN` est toutefois plus radical dans l'obscurcissement.

Incrustation

Le principe de fonctionnement de ce mode de fusion est plutôt complexe. La constante utilisée est `OVERLAY` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.OVERLAY;
```

Le résultat se trouve à la figure suivante.

Incrustation des couleurs

Proche de `HARDLIGHT`, ce mode de fusion est généralement très utilisé. Le rendu final est néanmoins légèrement différent de ce dernier dans les couleurs claires qui restent quasi-intactes.

Écran

La particularité de ce mode de fusion est de supprimer toute nuance de noir du calque. Il ne reste ensuite plus que la couleur « brute » de celui-ci. La constante qui le définit est `SCREEN` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.SCREEN;
```

Le résultat se trouve à la figure suivante.

Mode de fusion écran

Ce mode de fusion a donc pour effet d'éclaircir l'arrière-plan.

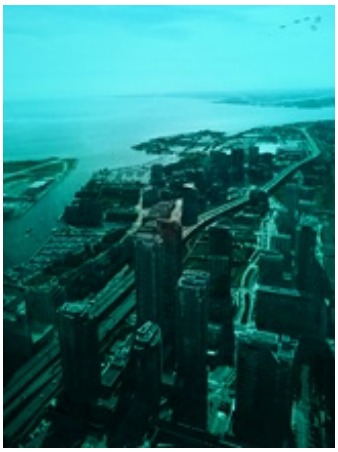
Soustraction

Comme vous vous en doutez, ce mode de fusion va soustraire les différents canaux de couleurs du calque à ceux de l'arrière-plan. Celui-ci correspond à la constante `SUBTRACT` :

Code : Actionscript

```
monCalque.blendMode = BlendMode.SUBTRACT;
```

Le résultat se trouve à la figure suivante.


Soustration des couleurs

Ce mode de fusion est donc l'opposé de `ADD`. Ainsi il peut être utilisé pour créer des transitions en utilisant cette fois un fondu d'assombrissement.

Les fusions de transparence

Transparence

Ce mode de fusion permet d'appliquer les niveaux de transparence du calque à l'arrière-plan. Comme nous l'avons dit, ce type de fusion nécessite de modifier également le mode de fusion du conteneur. Voici donc comment réaliser cette opération grâce à la constante `ALPHA` :

Code : Actionscript

```
this.blendMode = BlendMode.LAYER;
monCalque.blendMode = BlendMode.ALPHA;
```

Le résultat se trouve à la figure suivante.


Transfert de la transparence


Je rappelle qu'il existe une autre méthode qui consiste à définir des *masques* que nous introduirons dans le chapitre suivant. Personnellement je vous conseillerai plutôt d'utiliser les masques que la fusion de transparence.

Suppression

Le principe ici est l'opposé de `ALPHA`. Ainsi les niveaux de transparence du calque sont maintenant soustraits de l'arrière-plan. Le nom de la constante correspondant est donc `ERASE` :

Code : Actionscript

```
this.blendMode = BlendMode.LAYER;
monCalque.blendMode = BlendMode.ERASE;
```

Le résultat se trouve à la figure suivante.

Suppression de l'opacité

Ce mode de fusion est également moins courant. Évitez de l'utiliser si vous pouvez contourner le problème en ajustant l'ordre d'affichage des différents éléments graphiques.

En résumé

- Les **filtres** sont des effets qui sont applicable à tout objet graphique.
- Il existe *neuf* en Actionscript, regroupés dans le package `flash.filters`.
- Chaque filtre est associé à une classe, dont les propriétés permettent de décrire l'effet correspondant.
- Ceux-ci sont ensuite ajoutés à la propriété `filters` de type `Array` de tout objet `DisplayObject`.
- Les **modes de fusion** permettent de définir la manière dont plusieurs éléments doivent être fusionner lors de l'affichage final.
- Chaque élément héritant de `DisplayObject` dispose d'un unique mode de fusion défini dans la propriété `blendMode`.
- Tous ces modes de fusion sont définis à l'aide des constantes de la classe `BlendMode`.

Les masques

Pour le dernier chapitre théorique de cette partie sur l'affichage, je vous propose de découvrir les **masques** ! Ce concept qui permet de cacher certaines zones d'un objet affichage, est spécifique à l'Actionscript et à la technologie Flash. Pour ceux qui utilisent des logiciels de graphisme comme Gimp ou Photoshop, vous devriez vite cerner la chose. En revanche pour les autres, suivez attentivement ce que va être dit ici car cette technique est vraiment très pratique et très utilisée.

Un masque… qui ne masque pas

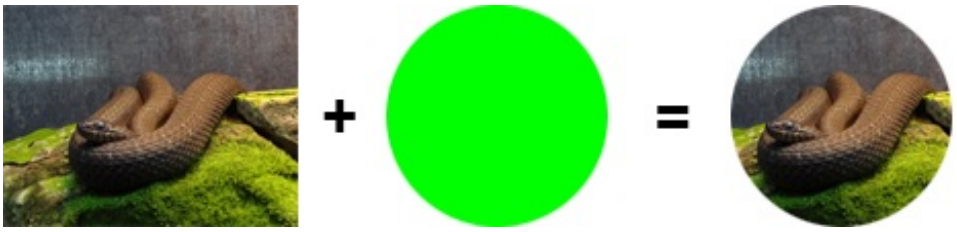
Le principe des masques

Lorsque vous créez des éléments graphiques, il est possible que vous ayez besoin d'isoler une certaine partie de l'objet d'affichage en question. C'est pourquoi vous aurez besoin de séparer la zone visible de la zone à cacher. Pour cela, nous utilisons ce qu'on appelle des **masques** !

Mais qu'est-ce qu'un masque ?

Un masque est un élément graphique comme un autre ; on retrouve alors des masques de type `Shape`, `Sprite` ou encore `Bitmap`. Celui-ci sert alors à délimiter la zone visible d'un autre objet d'affichage, définie par l'*opacité* du masque.

Pour bien comprendre, je vous propose à la figure suivante un petit schéma.



Principe d'utilisation des masques

Comme vous pouvez le voir, nous avons donc défini un masque circulaire, qui laisse apparaître la photo uniquement à l'intérieur de celui-ci. Bien évidemment, un masque peut être extrêmement complexe et composé de différentes formes. Par exemple, nous aurions pu suivre les courbes du serpent pour pouvoir l'isoler du reste de l'image.

Dans l'exemple précédent comme dans la suite du chapitre, j'utiliserai exclusivement des masques de couleur verte. Cela vous permettra de bien discerner les masques des autres objets d'affichage « classiques ».

Les masques en Flash

Définition du masque

Avant d'appliquer un masque à un objet visuel quelconque, il est nécessaire de définir ces deux éléments.
Pour commencer, je vous suggère un exemple relativement simple. Nous allons donc démarrer par deux objets `Shape` où nous dessinerons des formes basiques.

Dans ce chapitre, nous nous contenterons de définir des masques dessinés grâce aux méthodes de la classe `Graphics`. Toutefois, le principe avec des images « bitmap » est strictement identique. Il faudra néanmoins prendre en compte le *canal alpha*, comme nous le verrons dans la suite du chapitre.

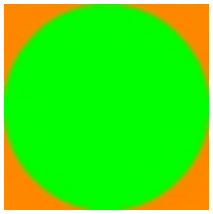
Voici donc nos deux éléments `monObjet` et `masque`, contenant respectivement un rectangle orange et un cercle vert :

Code : Actionscript

```
var monObjet:Shape = new Shape();
var masque:Shape = new Shape();
this.addChild(monObjet);
this.addChild(masque);

monObjet.graphics.beginFill(0xFF8800);
monObjet.graphics.drawRect(0, 0, 100, 100);
monObjet.graphics.endFill();
```

```
masque.graphics.beginFill(0x00FF00);
masque.graphics.drawCircle(50, 50, 50);
masque.graphics.endFill();
```

Nous avons ici placé l'objet `masque` au premier plan, ce qui donne la figure suivante.


L'objet et son masque


Lorsque vous définissez un masque avec les méthodes de la classe `Graphics`, seuls les remplissages sont utilisés pour définir un masque. Les lignes seront alors tout simplement ignorées.

L'objectif consiste donc maintenant à définir l'objet de couleur verte comme masque pour l'élément `monObjet`.

Une propriété `mask`

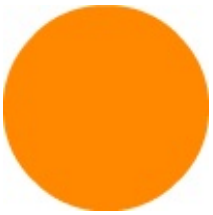
En fait la notion de masques est liée à tout objet visuel, c'est-à-dire tout objet de la classe `DisplayObject` ou plutôt d'une de ses sous-classes.
Nous avons déjà vu qu'une classe peut posséder un attribut d'elle-même ; la classe `DisplayObject` est justement l'une d'elles. En effet, celle-ci possède une propriété nommée `mask` de type `DisplayObject` !

Ainsi avec les objets de type `Shape` déclarés précédemment, il est possible de faire ceci :

Code : Actionscript

```
monObjet.mask = masque;
```

Dans cette instruction nous passons bien l'objet `masque` à la propriété `mask` de l'élément `monObjet`. Si vous relancez alors l'exécution du projet, vous verrez la magie s'opérer (voir figure suivante).


Effet du masque sur l'objet

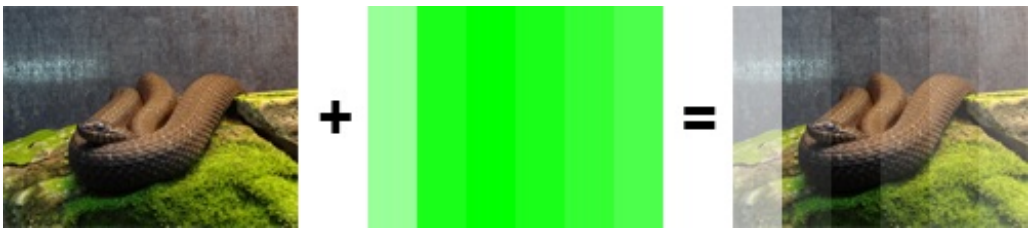

Comme vous pouvez le constater, le masque n'est maintenant plus visible. C'est pourquoi, l'instruction `this.addChild(masque)` est en réalité facultative. Toutefois si vous n'ajoutez pas le masque à la liste d'affichage, celui-ci ne sera alors pas affecté par les éventuelles transformations du conteneur. Pour éviter tout problème, je vous conseille donc de l'ajouter à chaque fois à la liste d'affichage.

Pour finir, veuillez noter qu'il est possible de supprimer le masque d'un objet simplement en définissant sa propriété `mask` à `null`.

Niveaux de transparence multiples

Présentation du concept

Comme nous avons vu précédemment, nous pouvons définir la forme d'un masque pour isoler certaines zones d'un objet d'affichage. Toutefois, il est également possible de définir la transparence de celui-ci en gérant l'opacité du masque. Rien ne nous empêche alors d'utiliser des **niveaux de transparence multiples**.
Pour illustrer ce qui vient d'être dit, je vous propose à la figure suivante un exemple de masque à plusieurs niveaux de transparence.



Principe des niveaux de transparence multiples

Comme le montre l'illustration précédente, l'objet d'affichage adapte donc sa transparence à l'opacité du masque qui lui est associé. Nous verrons alors qu'il est possible d'utiliser des dégradés pour définir un masque, ou bien utiliser directement une image « bitmap » dont le format supporte la transparence.

Place au code

Création du masque

Pour définir différents niveaux de transparence, il est nécessaire de « jouer » avec l'opacité des remplissages.
Je suggère alors que nous réalisions un masque composé de trois disques qui se superposent partiellement. Voici le code que je vous propose pour faire cela :

Code : Actionscript

```
// Objet à masquer
monObjet.graphics.beginFill(0xFF00FF);
monObjet.graphics.drawRect(0, 0, 160, 150);
monObjet.graphics.endFill();
// Masque
masque.graphics.beginFill(0x00FF00,0.5);
masque.graphics.drawCircle(80, 50, 50);
masque.graphics.endFill();
masque.graphics.beginFill(0x00FF00,0.5);
masque.graphics.drawCircle(50, 100, 50);
masque.graphics.endFill();
masque.graphics.beginFill(0x00FF00,0.5);
masque.graphics.drawCircle(110, 100, 50);
masque.graphics.endFill();
```

Nous avons donc ici réalisé un masque composé de trois niveaux de transparence différents, comme vous pouvez le voir à la figure suivante.

Un masque à plusieurs niveaux de transparence

Mise en place du masque

Redéfinissons maintenant la propriété `mask` de `monObjet` comme nous avons appris à le faire :

Code : Actionscript

```
monObjet.mask = masque;
```

Vous serez alors surpris de découvrir que ce masque ne fonctionne pas comme nous l'avions prévu, mais réagit comme si nous avions laissé une opacité maximale pour nos remplissages. Regardez plutôt la figure suivante.

Mauvaise application d'un masque à plusieur niveaux de transparence

En réalité pour pouvoir gérer la transparence d'un masque, nous devons utiliser une *mise en cache sous forme de bitmap*. L'élément graphique est alors géré comme s'il s'agissait d'une image « bitmap » en tenant compte du niveau de transparence de chacun de ses pixels.

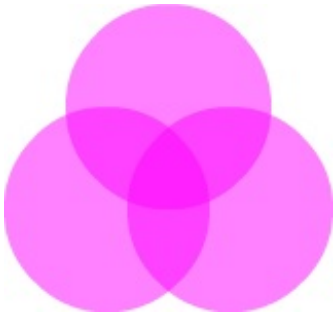
Le cache bitmap

Tout objet de type `DisplayObject` possède une propriété nommée `cacheAsBitmap`. Celle-ci permet d'activer la « mise en cache sous forme de bitmap », en la définissant tout simplement à **`true`**. Habituellement, les objets d'affichages sont manipulés par le lecteur Flash comme s'il s'agissait d'objets vectoriels, ce qui donne de bonnes performances la plupart du temps, mais implique quelques limitations, comme celle que nous venons de rencontrer. Lorsqu'un objet d'affichage est mis en cache sous forme de bitmap, il est d'abord rendu sous forme d'image contenant des pixels, puis manipulé par le lecteur Flash. Cela nous permet donc d'effectuer des opérations plus complexes (comme les masques à plusieurs niveaux de transparence). Ainsi, il nous faut activer ce cache sur le masque, mais également sur l'objet auquel il est appliqué.
Ainsi, voici comment procéder pour appliquer le masque à l'objet :

Code : Actionscript

```
monObjet.cacheAsBitmap = true;
masque.cacheAsBitmap = true;
monObjet.mask = masque;
```

Cette fois, vous verrez apparaître les différents niveaux de transparence, tels qu'ils ont été définis durant la création du masque. Nous obtenons la figure suivante.



Application correcte d'un masque à plusieurs niveaux de tranparence

Cette mise cache est également nécessaire lorsqu'on travaille avec des objets de type `Bitmap`. Le format d'image choisi doit néanmoins utiliser un canal alpha, comme c'est le cas des formats GIF ou PNG. La transparence est alors utilisée pour définir le masque à partir de cette image.

Mettre un objet d'affichage en cache bitmap a un impact sur les performances : ainsi, il est plus rapide de déplacer l'objet en cache bitmap, mais les transformations comme la rotation ou le redimensionnement ralentissent l'application de façon significative. Pour résumer, s'il vous voulez afficher un grand nombre d'objets qui se déplacent, vous pouvez améliorer les performances de votre application en activant la mise en cache bitmap, à condition de ne pas leur appliquer de transformations.

Exercice : une lampe torche

Pour terminer ce chapitre, nous allons réaliser un petit exercice ensemble.

Il s'agit d'un effet de lampe torche que nous créerons à partir de deux images et d'un masque. Cet exercice ne comporte aucune difficulté, contentez-vous uniquement de suivre ces différentes étapes avec moi.

Préparation des images

Rue de nuit

Pour commencer, je vous invite à créer un dossier nommé `images` où nous placerons nos deux images.
Dans un premier temps, nous aurons besoin d'une image d'une rue sombre qui servira de fond pour notre projet. Voici donc l'image que j'ai nommée « `RueSombre.png` » :

RueSombre.png

Cette image sera ainsi notre rue lorsqu'elle n'est pas éclairée par la lampe torche. Nous viendrons donc ajouter une partie lumineuse par dessus pour simuler l'effet d'une lampe torche.

Rue éclairée par la lampe

Lorsque notre lampe torche passera sur une zone, celle-ci sera éclairée presque comme en plein jour. Nous aurons alors besoin d'une seconde image de cette même rue, mais cette fois beaucoup plus lumineuse. Voici donc notre seconde image « `RueEclairee.png` » :

RueEclairee.png

La lampe torche n'agissant que sur une certaine partie de la rue, c'est cette image qui subira l'effet du masque que nous créerons juste après.

Chargement des images

En premier lieu, nous allons importer nos images dans notre animation Flash. Pour cela, Nous créerons deux variables de type `Class` que nous appellerons `RueSombre` et `RueEclairee`, comme vous avez appris à le faire.
Voici toutefois le code correspondant pour ceux qui auraient déjà oublié :

Code : Actionscript

```
// Chargement des images
[Embed(source = 'images/RueSombre.png')]
private var RueSombre:Class;
[Embed(source = 'images/RueEclairee.png')]
private var RueEclairee:Class;
```

N'oubliez pas d'ajouter les images dans le dossier de votre projet pour que celui-ci puisse compiler. Si jamais vous n'avez pas placé les images dans un dossier `images`, pensez alors à mettre à jour les liens ci-dessous.

Mise en place des images

Là encore, il n'y a aucune difficulté pour la mise en place des images. La seule chose à vérifier est de placer l'instance de la classe

`RueEclairee` *par dessus* celle de la classe `RueSombre`. Sinon vous ne verrez absolument rien dans le rendu final de l'animation.
Voici ce que je vous propose de faire :

Code : Actionscript

```
// Mise en place des images
var maRueSombre:Bitmap = new RueSombre();
var maRueEclairee:Bitmap = new RueEclairee();
this.addChild(maRueSombre);
this.addChild(maRueEclairee);
```

Mise en place du masque

À présent, nous allons nous occuper du masque que nous appliquerons à l'image de la rue éclairée. Cela devrait donc donner au final un effet de lampe torche, que vous pourrez déplacer pour éclairer la zone de la rue que vous souhaiterez.

Préparation du dégradé

La première étape est la préparation du dégradé qui nous servira à tracer notre masque.
Voici donc le code suggéré :

Code : Actionscript

```
// Préparation du dégradé
var couleurs:Array = [0x00FF00, 0x00FF00];
var alphas:Array = [1, 0];
var ratios:Array = [192, 255];
var matrix:Matrix = new Matrix();
matrix.createGradientBox(200, 200, 0, -100, -100);
```

Ce qu'il est important de noter, c'est que nous utiliserons une seule couleur où l'opacité diminuera sur les bords pour donner un effet d'atténuation de la lumière. Également nous voulons un dégradé rapide sur l'extérieur de la forme, c'est pourquoi nous avons augmenter la première valeur du tableau `ratios`.

Création de la lampe

Nous pouvons maintenant dessiner la forme du masque dans un objet de type `Sprite`. Nous allons donc définir un remplissage dégradé, puis nous tracerons un cercle pour représenter la forme du faisceau de lumière sortant de la lampe.
Voici comment réaliser ceci :

Code : Actionscript

```
// Création de la lampe
var lampe:Sprite = new Sprite();
lampe.graphics.beginGradientFill(GradientType.RADIAL, couleurs,
alphas, ratios, matrix);
lampe.graphics.drawCircle(0, 0, 100);
lampe.graphics.endFill();
this.addChild(lampe);
```

Voici donc à quoi ressemble le masque nous allons utiliser :

Aperçu du masque

Application du masque

Le masque réalisé précédemment va nous permettre d'afficher l'image de la rue éclairée uniquement dans la zone où la lampe pointe.
Pour faire cela, nous allons définir l'instance `lampe` comme masque pour l'objet `maRueEclairee`. Nous allons donc procéder comme nous avons appris à le faire :

Code : Actionscript

```
// Mise en place du masque
maRueEclairee.cacheAsBitmap = true;
lampe.cacheAsBitmap = true;
maRueEclairee.mask = lampe;
```

Nous avons à présent réaliser l'effet souhaité. Ceci dit, pour donner un peu plus d'intérêt à ce projet, nous allons animer celui-ci.

Animation de la lampe

Pour animer le masque, nous allons utiliser la méthode `startDrag()` de la classe `Sprite`. En indiquant son paramètre à **`true`**, on impose à l'objet en question de « coller » à la position de la souris. Ainsi, notre lampe va suivre les mouvements de la souris, pour imiter les gestes du poignet.
Voici donc comment procéder :

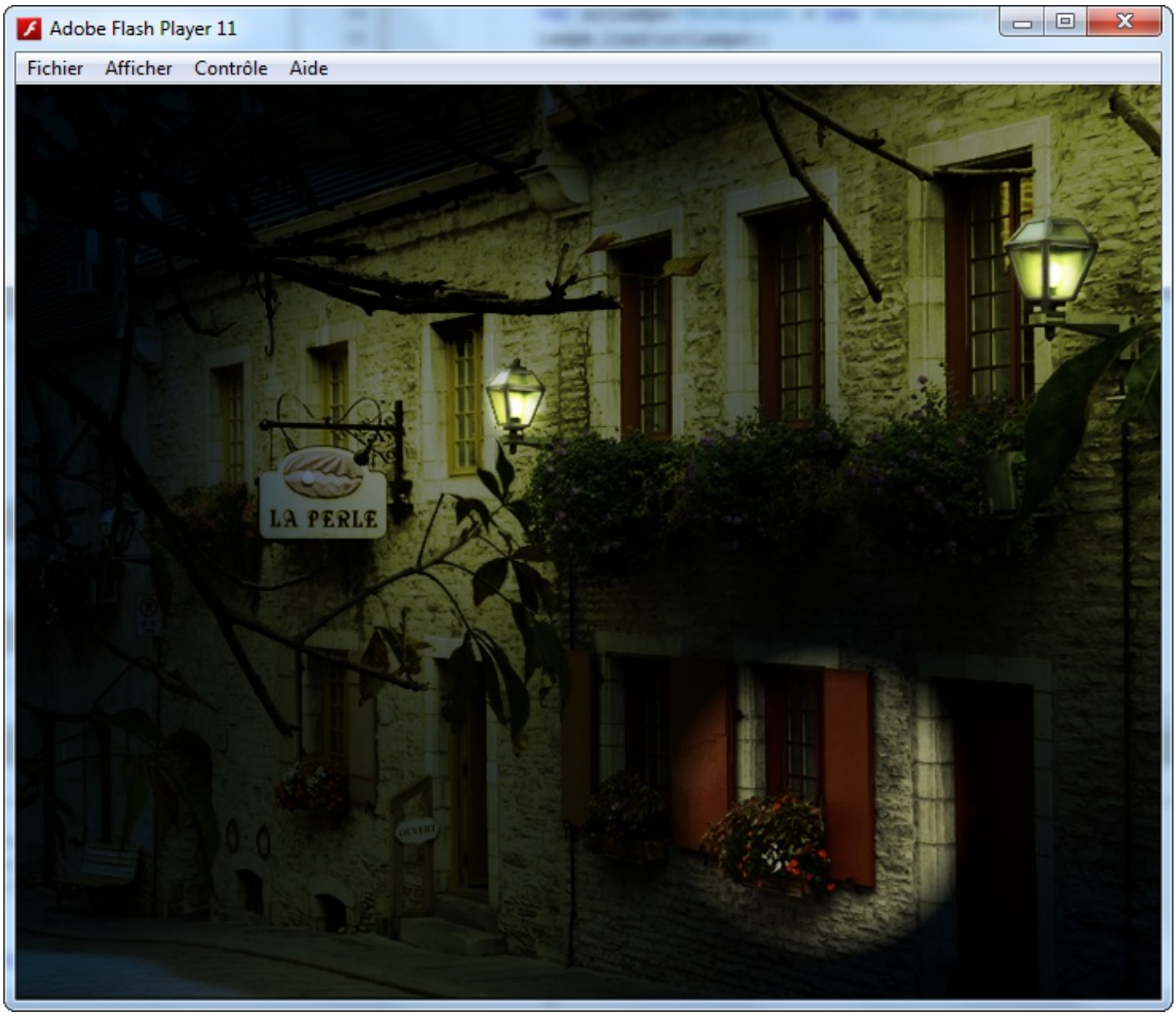
Code : Actionscript

```
// Animation de la lampe
lampe.startDrag(true);
```

Projet final

Le rendu

Nous avons maintenant terminer cet exercice sur les masques. Je vous propose donc ci-dessous un aperçu de l'animation finale :



Rendu final de l'exercice

Le code complet

Enfin pour finir, je vous ai ici recopier l'intégralité du code de ce projet :

Code : Actionscript

```
package
{
    import flash.display.Sprite;
    import flash.display.Shape;
    import flash.display.Bitmap;
    import flash.events.Event;
    import flash.geom.Matrix;
    import flash.display.GradientType;

    /**
* ...
* @author Guillaume
*/
    public class Main extends Sprite
    {
        // Chargement des images
        [Embed(source = 'images/RueSombre.png')]
        private var RueSombre:Class;
```

```
            private var RueSombre:Class;
            [Embed(source = 'images/RueEclairee.png')]
            private var RueEclairee:Class;

            public function Main():void
            {
                if (stage) init();
                else addEventListener(Event.ADDED_TO_STAGE, init);
            }

            private function init(e:Event = null):void
            {
                removeEventListener(Event.ADDED_TO_STAGE, init);

                // Mise en place des images
                var maRueSombre:Bitmap = new RueSombre();
                var maRueEclairee:Bitmap = new RueEclairee();
                this.addChild(maRueSombre);
                this.addChild(maRueEclairee);

                // Préparation du dégradé
                var couleurs:Array = [0x00FF00, 0x00FF00];
                var alphas:Array = [1, 0];
                var ratios:Array = [192, 255];
                var matrix:Matrix = new Matrix();
                matrix.createGradientBox(200, 200, 0, -100, -100);

                // Création de la lampe
                var lampe:Sprite = new Sprite();
                lampe.graphics.beginGradientFill(GradientType.RADIAL,
couleurs, alphas, ratios, matrix);
                lampe.graphics.drawCircle(0, 0, 100);
                lampe.graphics.endFill();
                this.addChild(lampe);

                // Mise en place du masque
                maRueEclairee.cacheAsBitmap = true;
                lampe.cacheAsBitmap = true;
                maRueEclairee.mask = lampe;

                // Animation de la lampe
                lampe.startDrag(true);
            }

        }

    }
```

En résumé

- Un **masque** est un objet d'affichage qui sert à délimiter une zone visible d'un autre objet.
- Lorsqu'on utilise la classe `Graphics`, seuls les remplissages sont pris en compte dans les masques.
- Le masque d'un objet se définit par sa propriété `mask`.
- Pour gérer les niveaux de transparence, la propriété `cacheAsBitmap` du masque et de l'objet doit être passée à **`true`**.
- Pour supprimer le masque d'un objet, sa propriété `mask` doit être redéfinie à la valeur **`null`**.

TP : Mauvais temps

Après ces trois longues parties plutôt théoriques, il est temps de passer un peu à la pratique !
Au cours de ce TP, vous aurez donc l'occasion de revenir sur l'ensemble des notions que vous avez vu jusqu'à présent. Étant donné la quantité de choses que vous avez dû accumuler jusqu'à maintenant, il est normal que vous ayez quelques difficultés. Ne soyez pas donc gêner d'aller relire un chapitre ou deux qui vous ont semblé flous à la première lecture. Toutefois vous êtes en mesure de réussir ce TP, c'est pourquoi je vous invite fortement à essayer par vous-mêmes avant de vous lancer directement dans la correction.

Le cahier des charges

L'objectif du chapitre

Le projet

L'objectif du chapitre est de concevoir une animation représentant une chute de neige !
Pour obtenir un rendu un peu plus réaliste, nous ajouterons également quelques petits trucs que je vous présenterais dans la suite. Avant d'en dire plus, je vous propose un petit aperçu de cette animation telle que nous aurons en fin de chapitre :

Aperçu de l'animation « Mauvais temps ».

Comme vous pouvez le voir, nous allons donc créer un certain nombre de flocons, que nous animerons afin de donner l'illusion d'une chute de neige. Nous ajouterons également une image de fond ainsi qu'un masque pour cacher les flocons derrière la branche au premier plan. D'autre part, les bords de l'image ont été assombris afin de mettre en valeur le centre de l'image.

Bien évidemment, le but pour vous est d'obtenir un rendu le plus proche possible du mien. Toutefois, il est impossible que vous obteniez exactement la même chose. L'objectif est de pratiquer et d'arriver à un résultat convenable, ne passez donc pas des jours à tenter d'obtenir des flocons quasi-identiques à ceux présentés juste au-dessus.

Également, il y a des dizaines de façons différentes de réaliser une telle animation. C'est pourquoi je vous présenterai ici la manière que j'ai choisie, mais n'ayez pas peur si vous avez une méthode différente pour aborder le problème.
Avant de vous lancer tête baissée, lisez bien toutes les consignes et prérequis, sans quoi vous n'arriverez certainement pas au bout !

Intérêt du projet

Ce projet a été choisi pour diverses raisons que voici :

- Tout d'abord, il s'agit d'une animation très répandu en Flash, et que beaucoup de débutants désirent réaliser.
- Dans ce projet, nous allons utiliser la quasi-totalité des connaissances acquises jusqu'à présent à savoir: *variables*,

conditions, *boucles*, *fonctions*, *tableaux*, *classes* et *programmation orientée objet*, et quasiment tout ce qui est lié à l'*affichage*.
- Enfin, cette animation ne nécessite pas énormément de lignes de code, ce qui est plutôt favorable pour l'un de vos premiers TPs.

Bien entendu, nous allons détailler un peu plus la manière de procéder pour concevoir ce projet.

Le travail à réaliser

Pour réaliser cette animation, je vous invite à procéder petit à petit. C'est pourquoi je vous ai ici présenter les différentes étapes de la conception de ce projet.

Création de l'aspect d'un flocon

La première étape va consister à dessiner un flocon. Vous avez déjà toutes les connaissances pour parvenir à dessiner celui-ci, vous devriez donc y arriver sans problème. Il est possible de réaliser des formes de flocons extrêmement complexes, néanmoins un simple disque fera très bien l'affaire.
Le fait qu'il y aura des milliers de flocons dans notre futur animation devrait vous faire naître des idées quant à la manière de concevoir ceux-ci. Je n'en dis pas plus, à vous de réfléchir d'ores et déjà à la façon de faire !

La génération des flocons

Ensuite vous allez devoir générer l'ensemble de vos flocons. Vous allez donc générer un certain nombre de flocons que nous réutiliserons à l'infini. Une fois généré, chaque flocon devra être positionné aléatoirement à l'écran. Il vous faudra également définir aléatoirement son aspect ainsi que sa vitesse de chute. Pensez également que vous pouvez donner de la profondeur à votre scène, sachant que les flocons les plus gros seront au premier plan et les plus rapides de notre point de vue.

L'animation des flocons

Tout ce qui concerne l'animation sera regroupé à l'intérieur d'une méthode qui sera appelée à intervalles réguliers grâce à la fonction `setInterval()`.
À l'intérieur de cette méthode, vous ferez l'ensemble des modifications nécessaires à l'animation des flocons. Vous pourrez ainsi mettre à jour la position de chaque flocon, et éventuellement réinitialiser un flocon sorti de la scène. Cette fonction sera donc appelée régulièrement et à l'infini, et se chargera ainsi de l'animation de la scène.

C'est un peu prématuré puisqu'il s'agit de l'objet du prochain chapitre, mais vous utiliserez pour cela ce qu'on appelle une **fonction de rappel**. Sans plus d'explications, voici comment utiliser cette fonction `setInterval()` :

Code : Actionscript

```
var monInterval:uint = setInterval(maMethode, intervalle);
```

Dans l'instruction précédente, `maMethode` représente une méthode, donc de type `Function`, et `intervalle` correspond au temps espaçant deux exécutions consécutives de la fonction précédente.

Association des différents éléments graphiques

Lorsque que toute la gestion des flocons aura été terminée, vous pourrez alors passer à la composition de la scène finale. Vous devrez donc ajouter l'image de fond et le masque à la scène. Vous pourrez également tenter de créer cet effet d'assombrissement des bords de la scène en étant imaginatif.
Cette étape devrait normalement se dérouler relativement bien si vous être parvenu jusque là !

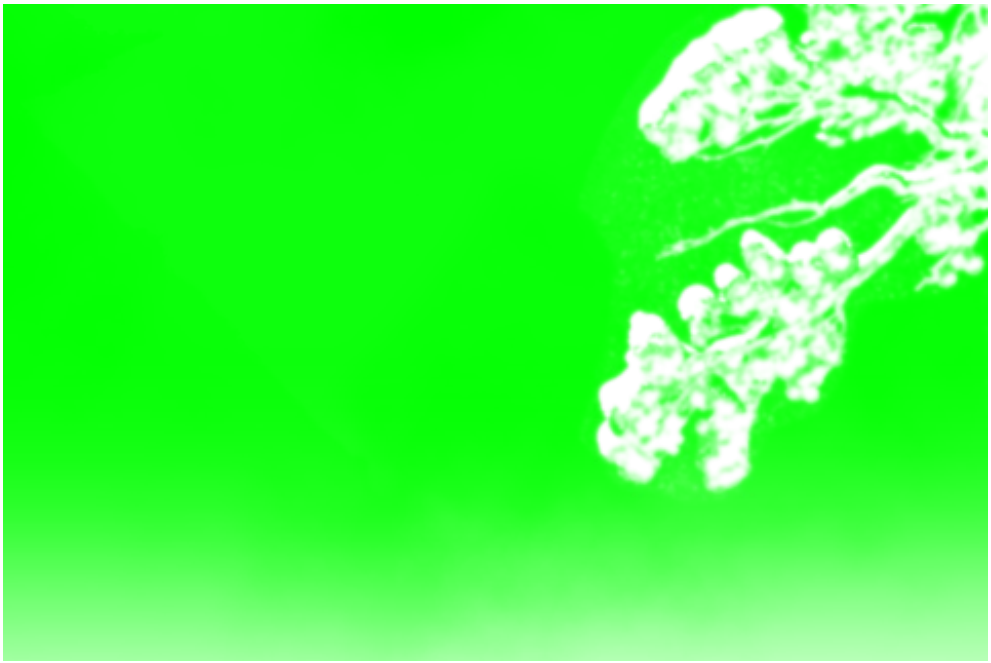
Les images

Enfin je vous suggère de télécharger les deux images dont vous aurez besoin pour réaliser ce TP. Il y a donc l'image de fond intitulée « `Paysage.png` » ainsi que le masque nommé « `Masque.png` ».

L'image de fond

Paysage.png

Le masque

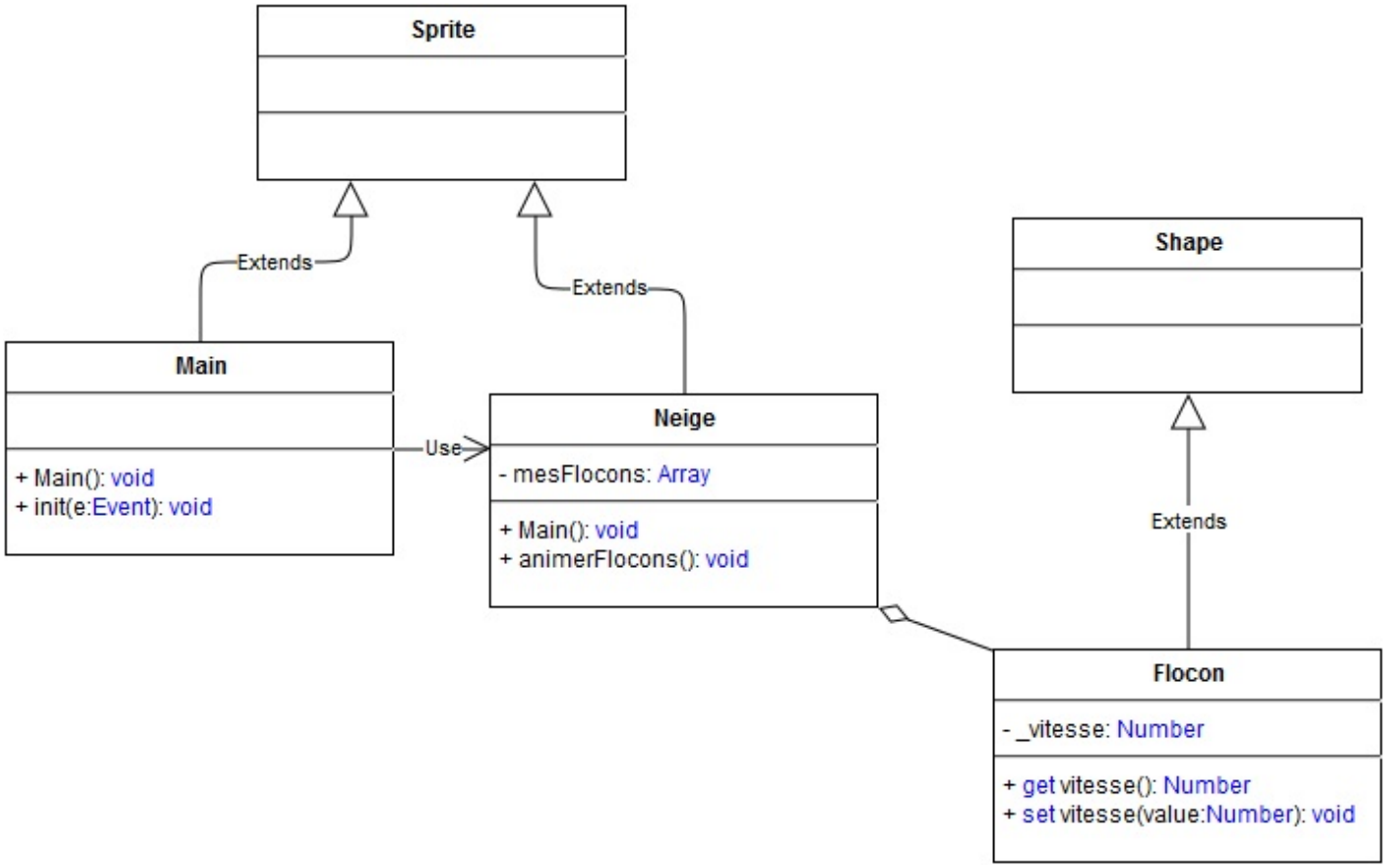


Masque.png

La correction

La structure du programme

Avant de nous lancer tête baissée dans les lignes de code, je vais vous présenter un peu la manière dont j'ai conçu et organiser le code au sein du projet. Si vous avez réaliser ce projet par vous-mêmes, il est probable que vous n'ayez pas structuré votre projet de la même façon ; je rappelle qu'il n'y a pas qu'une unique manière de concevoir une telle animation.
Voici donc comment j'ai procédé pour programmer ce TP :



La structure du projet telle qu'elle est définie dans la correction.

Ainsi dans l'organisation précédente, nous avons divisé le code en trois classes différentes :

- La classe `Flocon` est celle qui va principalement vous permettre de dessiner un flocon. Elle hérite donc de `Shape` pour avoir accès aux méthodes de la classe `Graphics`. Également vous pouvez associer à cette classe un attribut `_vitesse` qui correspond à la vitesse de chute du flocon et qui lui est donc propre.
- Ensuite nous avons la classe `Neige` qui hérite de `Sprite` et est donc un conteneur. Elle va donc nous permettre de générer et d'animer l'ensemble des flocons.
- Enfin, notre classe principale va nous servir à composer la scène finale à l'aide de l'image de fond, du masque et d'un dernier objet que nous créerons et qui servira à assombrir les bords de la scène.

Dans la suite, nous allons donc détailler l'ensemble de ses classes, en partant de la classe `Flocon` pour aller vers la classe principale.

La classe `Flocon`

Création de la classe

La première classe que nous allons créer est la classe `Flocon` !
Voici donc la structure de base de cette classe :

Code : Actionscript

```
package
{
    import flash.display.Shape;

    public class Flocon extends Shape
    {
        public function Flocon()
        {
            super();

        }

    }
```

```
    }
```

Celle-ci permet tout d'abord de créer le « visuel » du flocon. Pour cela, nous utiliserons les méthodes de la classe `Graphics` puisque notre classe hérite de `Shape`. Rien de plus facile maintenant pour vous, il suffit de tracer un disque blanc à l'intérieur du constructeur de la manière suivante :

Code : Actionscript

```
this.graphics.beginFill(0xFFFFFF);
this.graphics.drawCircle(0, 0, 1.5);
```

Sachant que nos flocons de tomberont pas tous à la même vitesse, nous allons donc créer un attribut `_vitesse` pour retenir celle-ci :

Code : Actionscript

```
private var _vitesse:Number;
```

Enfin, n'oubliez pas d'initialiser cet attribut et de lui associer des accesseurs, que nous ne détaillerons pas ici. Jetez simplement un coup d'œil au code complet de cette classe, qui entre nous est plutôt succinct.

La classe Neige

Création de la classe

À présent, nous allons nous occuper de la classe `Neige` qui sera donc notre conteneur pour l'ensemble des flocons.
Pour commencer, je vous invite donc à créer une nouvelle classe si ce n'est pas déjà fait. À l'intérieur, nous aurons besoin des trois constantes que j'ai nommé `LARGEUR_SCENE`, `HAUTEUR_SCENE` et `NOMBRE_FLOCONS`. Pour faciliter la suite, j'ai déjà fait l'intégralité des importations ainsi que la mise en place de la structure globale de la classe.
Voici donc notre classe `Neige`, prête à être remplie :

Code : Actionscript

```
package
{
    import flash.display.Sprite;
    import flash.filters.BlurFilter;
    import flash.filters.BitmapFilterQuality;
    import flash.utils.*;

    public class Neige extends Sprite
    {
        private const LARGEUR_SCENE:int = 480;
        private const HAUTEUR_SCENE:int = 320;
        private const NOMBRE_FLOCONS:int = 3000;

        public function Neige()
        {
            super();
            // Génération des flocons
        }

        public function animerFlocons():void
        {
            // Animation des flocons
        }
    }

}
```

Génération des flocons

Nous allons ici nous intéresser à la création des flocons et à leur mise en place sur la scène au démarrage de l'animation.
Pour commencer, nous allons créer nos flocons en utilisant un tableau de `Flocon` de type `Array`. Ce tableau devra être accessible depuis la fonction `animerFlocons()`, c'est pourquoi j'ai décidé de créer un nouvel attribut nommé `MesFlocons` :

Code : Actionscript

```
// Tableau de Flocon
private var mesFlocons:Array;
```

Dans ce paragraphe, nous nous occuperons de la génération des flocons. Plaçons-nous donc à l'intérieur du constructeur de la classe `Neige`.
Commençons tout de suite par initialiser notre tableau `Array`, en définissant sa taille à `NOMBRE_FLOCONS` défini juste avant :

Code : Actionscript

```
mesFlocons = new Array();
```

À présent, nous devons instancier nos flocons et leur apporter quelques modifications *un par un*. Il se trouve que les boucles sont justement très recommandées pour parcourir les éléments d'un tableau :

Code : Actionscript

```
for (var i:uint = 0; i < NOMBRE_FLOCONS - 1; i++) {
    // Création et modification de chaque flocon
}
```

Nous pouvons maintenant instancier nos flocons, les uns après les autres :

Code : Actionscript

```
mesFlocons.push(new Flocon());
```

Lors du démarrage de l'animation, il serait préférable que les flocons soient répartis *aléatoirement* sur l'écran. Nous allons donc modifier la position horizontale et la position verticale en utilisant la méthode `random()` de la classe `Math`.
Voici comment réaliser cette opération :

Code : Actionscript

```
// Placement aléatoire des flocons sur l'écran
mesFlocons[i].x = Math.random() * LARGEUR_SCENE;
mesFlocons[i].y = Math.random() * HAUTEUR_SCENE;
```

Pour donner une impression de profondeur à notre scène, nous allons modifier l'aspect de nos flocons ainsi que leur vitesse.
Pour cela nous allons définir une variable `aspect` qui représentera la taille du flocon, mais également sa « distance » à l'écran ; le zéro correspondant à une distance infinie, c'est-à-dire loin de l'écran. Étant donné qu'avec la distance, la vitesse de chute des flocons semble diminuer également, nous réutiliserons cette variable pour définir la vitesse du flocon.
Voici le code correspondant à ces manipulations :

Code : Actionscript

```
// Définition d'un aspect aléatoire
```

```
var aspect:Number = Math.random() * 1.5;
// Modification de l'aspect du flocon
mesFlocons[i].scaleX = aspect;
mesFlocons[i].scaleY = aspect;
mesFlocons[i].alpha = aspect;
// Modification de la vitesse du flocon
mesFlocons[i].vitesse = 2 * (aspect + Math.random());
```

Suivant la distance du flocon à l'écran, celui-ci sera également plus ou moins net. Nous allons donc réutiliser notre variable `aspect` pour créer un filtre de flou, simulant la profondeur de champ de la caméra. Pour accentuer cet effet de profondeur, j'ai utilisé une formule mathématique avec des puissances.
Je vous donne directement le code correspondant :

Code : Actionscript

```
// Génération du filtre
var flou:BlurFilter = new BlurFilter();
flou.blurX = Math.pow(10*aspect,4)/10000;
flou.blurY = Math.pow(10*aspect,4)/10000;
flou.quality = BitmapFilterQuality.MEDIUM;
// Application du filtre au flocon
mesFlocons[i].filters = new Array(flou);
```

Enfin, il ne nous reste plus qu'à ajouter notre flocon à la liste d'affichage de la classe `Neige` :

Code : Actionscript

```
this.addChild(mesFlocons[i]);
```

La génération des flocons est maintenant terminée, et notre application est prête à être animée ! 
Pour cela, n'oubliez pas de mettre l'instruction suivante à l'extérieur de la boucle :

Code : Actionscript

```
var monInterval:uint = setInterval(animerFlocons, 40);
```

L'animation des flocons

Dans ce paragraphe, nous allons nous occuper de toute la partie « animation » du programme. Nous travaillerons donc à l'intérieur de la méthode `animerFlocons()` qui, je le rappelle, sera exécutée toutes les 40 millisecondes grâce à la fonction `setInterval()`.
Encore une fois, nous allons manipuler les flocons individuellement. Nous utiliserons donc à nouveau une boucle **`for`** :

Code : Actionscript

```
for (var i:uint = 0; i < mesFlocons.length; i++) {
    // Mise à jour de chaque flocon
}
```

Le principe de l'animation consiste à modifier la position de chaque flocon. Pour faire cela, nous utiliserons la propriété `vitesse` de la classe `Flocon` que nous viendrons ajouter à la position verticale du flocon.
Voici comment il faut procéder :

Code : Actionscript

```
mesFlocons[i].y += mesFlocons[i].vitesse;
```

Pour éviter de créer des flocons à l'infini, nous allons réutiliser les flocons qui sont sortis de la zone visible de l'écran pour les renvoyer en haut de celui-ci. Vous vous en doutez peut-être ; nous utiliserons une condition **if**, en testant la valeur de la position verticale d'un flocon comme ceci :

Code : Actionscript

```
if (mesFlocons[i].y > HAUTEUR_SCENE) {
    // Réinitialisation du flocon en haut de la scène.
}
```

À l'intérieur, nous pouvons alors remettre la position verticale du flocon à zéro et éventuellement replacer aléatoirement le flocon horizontalement :

Code : Actionscript

```
mesFlocons[i].x = Math.random() * LARGEUR_SCENE;
mesFlocons[i].y = 0;
```

Aussi surprenant que cela puisse paraître, nous avons à présent fini l'animation des flocons !

La classe principale

Création des objets d'affichage

Nous voici de retour dans la classe principale !
Nous allons donc ici gérer la mise en place des différents éléments à l'écran. Dans un premier temps, je vous propose d'importer les deux images externes, à savoir `Paysage.png` et `Masque.png` que vous avez dû normalement télécharger précédemment.
Voici comment j'ai réalisé ceci :

Code : Actionscript

```
// Préparation des images
[Embed(source = 'images/Paysage.png')]
private var Paysage:Class;
[Embed(source = 'images/Masque.png')]
private var Masque:Class;
```

Au final, j'ai utilisé quatre éléments différents pour composer la scène finale. J'ai tout d'abord pris deux occurrences `monPaysage` et `monMasque` des images précédentes. Ensuite j'ai instancié notre classe `Neige` qui contient l'ensemble des flocons de l'animation. Enfin pour terminer, j'ai créer une nouvelle variable nommée `maVignette` de type `Shape`. Celle-ci va nous permettre de créer un « effet de vignette », comme nous le verrons juste après.
Voilà donc nos quatre objets :

Code : Actionscript

```
// Création des objets d'affichage
var monPaysage:Bitmap = new Paysage();
var monMasque:Bitmap = new Masque();
var maNeige:Neige = new Neige();
var maVignette:Shape = new Shape();
```

Dans la suite, nous allons alors gérer l'« interconnexion » entre ces différents éléments pour aboutir au rendu final que vous avez vu en début de chapitre.

Création d'un effet de vignette

Un « effet de vignette » est une technique qui permet de mettre en valeur ce qui se trouve au centre d'une image. En fait, cela consiste à assombrir les bords et les angles de l'image. Pour faire ça, le plus simple est créer un dégradé elliptique, de couleur blanche au centre et sombre à l'extérieur. Ainsi par application d'un mode de fusion, le centre de l'image se voit devenir plus lumineuse et ressort davantage.
Voici donc le vignettage que nous allons réaliser :

Dégradé radial pour un « effet de vignette »

Pour réaliser cet effet, nous allons créer un dégradé de type radial à l'intérieur d'un remplissage de forme rectangulaire.
Pour commencer, nous nous occuperons des réglages du dégradé en termes de couleurs. Nous aurons donc un dégradé du blanc au noir, comme vous pouvez le voir ci-dessous :

Code : Actionscript

```
// Paramètrage du dégradé
var couleurs:Array = [0xFFFFFF, 0x000000];
var alphas:Array = [1, 1];
var ratios:Array = [96, 255];
```

Ensuite, il nous faut générer la matrice permettant de contrôler l'aspect général du dégradé. Pour cela, nous allons donc créer un dégradé radial plus large que la taille de la scène afin de ne pas avoir de noir absolu, puis nous recentrerons celui-ci.
Voici comment j'ai réalisé ceci :

Code : Actionscript

```
// Création de la matrice de description
var matrix:Matrix = new Matrix();
var largeur:Number = 2*LARGEUR_SCENE;
var hauteur:Number = 2*HAUTEUR_SCENE;
var rotation:Number = 0;
var tx:Number = -0.5*LARGEUR_SCENE;
var ty:Number = -0.5*HAUTEUR_SCENE;
matrix.createGradientBox(largeur, hauteur, rotation, tx, ty);
```

L'étape suivante consiste alors à tracer le remplissage correspondant à la taille de la scène :

Code : Actionscript

```
// Tracé du remplissage
maVignette.graphics.beginGradientFill(GradientType.RADIAL, couleurs,
alphas, ratios, matrix);
maVignette.graphics.drawRect(0, 0, LARGEUR_SCENE, HAUTEUR_SCENE);
```

Enfin pour finir, nous allons appliquer un mode de fusion de type « multiplier » qui va permettre d'assombrir les bords de l'image en laissant le centre de celle-ci intact.

Code : Actionscript

```
// Application du mode de fusion
maVignette.blendMode = BlendMode.MULTIPLY;
```

Nous avons à présent l'intégralité de nos éléments graphiques qu'il va falloir agencer les uns avec les autres pour donner le rendu final.

Mise en place des éléments à l'écran

Maintenant, nous devons ajouter chacun de nos éléments à la liste d'affichage. Mise à part le masque qui ne sera pas visible, les objets d'affichage doivent être ajoutés dans un certain ordre. En arrière-plan, nous aurons donc l'image de fond présente dans la variable `monPaysage`. Ensuite nous placerons l'objet `maNeige` par-dessus qui sera masqué par `monMasque`. Enfin, nous placerons `maVignette` au premier plan pour que son mode de fusion s'applique à tous les objets derrière elle.
Voilà donc l'ordre à respecter :

Code : Actionscript

```
// Ajout des éléments à la liste d'affichage
this.addChild(monPaysage);
this.addChild(monMasque);
this.addChild(maNeige);
this.addChild(maVignette);
```

Pour finir, appliquez le masque à l'objet `maNeige` :

Code : Actionscript

```
// Application du masque
maNeige.cacheAsBitmap = true;
monMasque.cacheAsBitmap = true;
maNeige.mask = monMasque;
```

Ça y est, c'est terminé ! 
Nous avons à présent fini l'intégralité de cette animation qui finalement demandait bien quelques efforts.

Le code source complet

La classe `Flocon`

Code : Actionscript - Flocon.as

```
package
{
    import flash.display.Shape;

    public class Flocon extends Shape
    {
        private var _vitesse:Number;

        public function Flocon()
        {
```

```
                super();
                _vitesse = 0;
                this.graphics.beginFill(0xFFFFFF);
                this.graphics.drawCircle(0, 0, 1.5);
            }

            public function get vitesse():Number
            {
                return _vitesse;
            }

            public function set vitesse(value:Number):void
            {
                _vitesse = value;
            }

        }

    }
```

La classe Neige

Code : Actionscript - Neige.as

```
    package
    {
        import flash.display.Sprite;
        import flash.filters.BlurFilter;
        import flash.filters.BitmapFilterQuality;
        import flash.utils.*;

        public class Neige extends Sprite
        {
            private const LARGEUR_SCENE:int = 480;
            private const HAUTEUR_SCENE:int = 320;
            private const NOMBRE_FLOCONS:int = 3000;
            // Tableau de Flocon
            private var mesFlocons:Array;

            public function Neige()
            {
                super();
                mesFlocons = new Array();
                for (var i:uint = 0; i < NOMBRE_FLOCONS - 1; i++) {
                    // Création et modification de chaque flocon
                    mesFlocons.push(new Flocon());
                    // Placement aléatoire des flocons sur l'écran
                    mesFlocons[i].x = Math.random() * LARGEUR_SCENE;
                    mesFlocons[i].y = Math.random() * HAUTEUR_SCENE;
                    // Définition d'un aspect aléatoire
                    var aspect:Number = Math.random() * 1.5;
                    // Modification de l'aspect du flocon
                    mesFlocons[i].scaleX = aspect;
                    mesFlocons[i].scaleY = aspect;
                    mesFlocons[i].alpha = aspect;
                    // Modification de la vitesse du flocon
                    mesFlocons[i].vitesse = 2 * (aspect +
    Math.random());
                    // Génération du filtre
                    var flou:BlurFilter = new BlurFilter();
                    flou.blurX = Math.pow(10*aspect,4)/10000;
                    flou.blurY = Math.pow(10*aspect,4)/10000;
                    flou.quality = BitmapFilterQuality.MEDIUM;
                    // Application du filtre au flocon
                    mesFlocons[i].filters = new Array(flou);
                    // Ajout du flocon à la liste d'affichage
```

```
                this.addChild(mesFlocons[i]);
            }

            var monInterval:uint = setInterval(animerFlocons, 25);
        }

        public function animerFlocons():void
        {
            for (var i:uint = 0; i < mesFlocons.length; i++) {
                // Mise à jour de chaque flocon
                mesFlocons[i].y += mesFlocons[i].vitesse;
                if (mesFlocons[i].y > HAUTEUR_SCENE) {
                    // Réinitialisation du flocon en haut de la
scène.
                    mesFlocons[i].x = Math.random() * LARGEUR_SCENE;
                    mesFlocons[i].y = 0;
                }
            }
        }
    }

}
```

La classe principale

Code : Actionscript - Main.as

```
package
{
    import flash.display.Shape;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.display.Bitmap;
    import flash.geom.Matrix;
    import flash.display.GradientType;
    import flash.display.BlendMode;

    [SWF(width="480", height="320")]
    public class Main extends Sprite
    {
        private const LARGEUR_SCENE:int = 480;
        private const HAUTEUR_SCENE:int = 320;
        // Préparation des images
        [Embed(source = 'images/Paysage.png')]
        private var Paysage:Class;
        [Embed(source = 'images/Masque.png')]
        private var Masque:Class;

        public function Main():void
        {
            if (stage) init();
            else addEventListener(Event.ADDED_TO_STAGE, init);
        }

        private function init(e:Event = null):void
        {
            removeEventListener(Event.ADDED_TO_STAGE, init);
            // Création des objets d'affichage
            var monPaysage:Bitmap = new Paysage();
            var monMasque:Bitmap = new Masque();
            var maNeige:Neige = new Neige();
            var maVignette:Shape = new Shape();
            // Paramétrage du dégradé
            var couleurs:Array = [0xFFFFFF, 0x000000];
            var alphas:Array = [1, 1];
            var ratios:Array = [96, 255];
```

```
                // Création de la matrice de description
                var matrix:Matrix = new Matrix();
                var largeur:Number = 2*LARGEUR_SCENE;
                var hauteur:Number = 2*HAUTEUR_SCENE;
                var rotation:Number = 0;
                var tx:Number = -0.5*LARGEUR_SCENE;
                var ty:Number = -0.5*HAUTEUR_SCENE;
                matrix.createGradientBox(largeur, hauteur, rotation, tx,
ty);
                // Tracé du remplissage

maVignette.graphics.beginGradientFill(GradientType.RADIAL, couleurs,
alphas, ratios, matrix);
                maVignette.graphics.drawRect(0, 0, LARGEUR_SCENE,
HAUTEUR_SCENE);
                Application du mode de fusion
                maVignette.blendMode = BlendMode.MULTIPLY;
                // Ajout des éléments à la liste d'affichage
                this.addChild(monPaysage);
                this.addChild(monMasque);
                this.addChild(maNeige);
                this.addChild(maVignette);
                // Application du masque
                maNeige.cacheAsBitmap = true;
                monMasque.cacheAsBitmap = true;
                maNeige.mask = monMasque;

            }

        }

    }
```

Partie 4 : Interaction et animation

Les événements

Nous voici dans une nouvelle partie, portant sur l'interactivité et l'animation en Actionscript. Ces deux concepts sont étroitement liés à la notion d'évènements, qui est l'objet de ce chapitre.
Ce chapitre d'introduction sera malheureusement le plus théorique et le plus complexe de cette partie. Une fois que vous aurez acquis les bases de la programmation événementielle, les autres chapitres seront beaucoup plus pratiques et faciles à aborder.

Qu'est ce qu'un évènement ?
Introduction

Dans ce chapitre nous allons aborder la programmation événementielle. A l'instar la POO, cette manière de programmer s'inspire d'éléments du quotidien, à savoir : les **évènements**.

Attention, j'en vois déjà certains penser aux festivals de cet été : il ne s'agit pas de ça ! 
Les évènements, tels que nous allons les voir, sont définis comme des éléments non prévisibles, généralement déclenché par un changement d'état quelconque. Par exemple dans la vie courante, nous pouvons citer :

- un réveil qui sonne,
- un accident de voiture,
- une panne de courant,
- la réception d'un fax.

Pour compléter la définition précédente, nous pourrions dire qu'il s'agit d'évènements soudains qui ne surgisse pas vraiment lorsqu'on s'y attend. Dans l'exemple du réveil, il est vrai qu'on s'attend forcément à ce qu'il sonne, cependant vous devez avouer que vous êtes toujours surpris lorsque ce dernier se met à sonner !

En programmation, le principe est le même. Dans le cas de l'ActionScript 3, nous pourrions citer comme événements :

- un clic à l'aide de la souris,
- le déplacement du curseur de la souris,
- l'enfoncement d'une touche du clavier,
- la fin de chargement d'une image depuis un serveur,
- une erreur lors du chargement de cette image.

À l'apparition de l'un des évènements, il est alors possible d'exécuter des instructions spécifiques, pour mettre à jour l'affichage par exemple. Étant donné qu'on ne sait pas à l'avance quand ceux-ci se produiront, il serait possible mais très fastidieux d'inclure des instructions au milieu de votre programme qui bouclent indéfiniment pour détecter ces événements : c'était la méthode avant de l'entrée en scène du concept de **gestion des évènements**, qui facilite grandement ce processus.

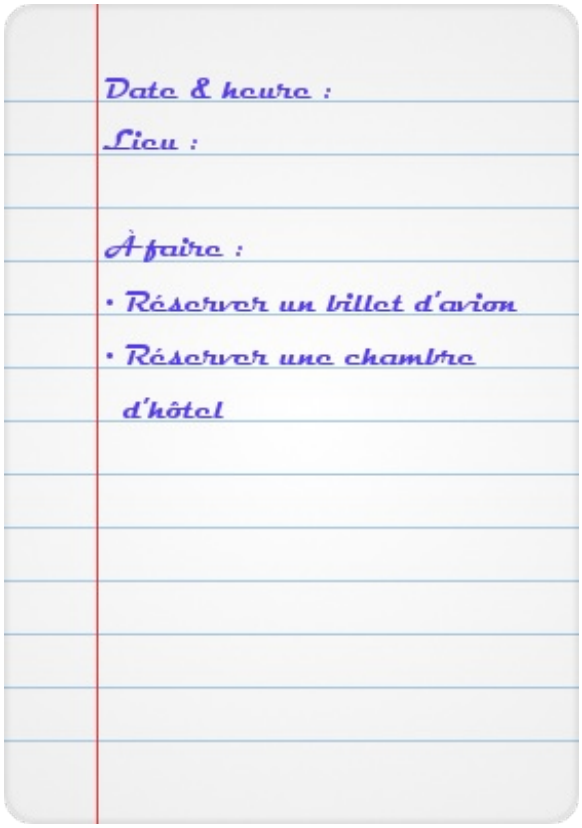
Pour introduire ce concept, nous allons maintenant faire une analogie en prenant l'idée de la réception d'un fax.

Un jour au bureau...

Aujourd'hui, vous travaillez pour une grande entreprise de renommée internationale. Votre compagnie dispose ainsi de différentes agences réparties sur plusieurs continents. En tant que responsable produits, vous allez devoir prochainement présenter un nouveau concept à l'ensemble du groupe. Il est donc prévu que vous assistiez à une réunion avec des collaborateurs de diverses agences. En revanche, les détails pratiques tels que l'heure et l'endroit de cette réunion ne sont pas encore fixés.

Fonction d'écouteur

Monsieur Gumble, directeur d'une agence nord-américaine, a promis qu'il vous enverrait les informations concernant la réunion aujourd'hui par fax. Comme à votre habitude, ce matin-là vous arrivez au bureau légèrement en avance. Vous en profitez alors pour sortir une feuille, et commencez à noter différentes choses dessus :



Une note que vous avez griffonnée.

Machinalement, vous avez commencé à anticiper et vous avez noté les différentes tâches à faire à la réception de ce fameux fax : cela ressemble à une fonction contenant une série d'instructions, non ? Cette « fonction » qui devra être appelée dès que vous recevrez le fax est désignée comme étant une **fonction d'écouteur**.

En ActionScript 3, elle sera donc une fonction quelconque ou une méthode de classe regroupant des instructions à exécuter lors de l'apparition d'un **évènement**. Généralement, les actions qui en découleront, dépendent de paramètres liés à l'évènement. Par exemple, savoir quelle touche a été enfoncée permettrait d'exécuter des instructions différentes suivant s'il s'agit d'un A ou d'un S.

Écouter un événement

Ça y est ! Vous êtes prêts à recevoir votre fax. Vous jetez régulièrement un coup d'œil sur le télécopieur qui se trouve dans le coin :

Le télécopieur est la cible de l'événement « recevoir un fax ».

Attendre qu'un événement survienne est appelé **écouter un événement**. On écoute toujours sur un objet désigné comme étant la

cible de l'événement. On dit également que cet objet **déclenchera l'événement** lorsque celui-ci surviendra.
Ici, le télécopieur est la cible de l'événement qui se produira à la réception d'un fax.

En programmation, on écoute un événement particulier en renseignant une sorte de « signature », appelée **type d'événement**, décrite par une chaîne de caractères : il est alors possible de distinguer un clic de souris du déplacement du curseur au-dessus d'un objet identique.
Écouter un événement qu'un objet sera susceptible de déclencher consiste donc à *associer* une fonction d'écouteur à un type d'évènements tout en ciblant cet objet. Ainsi, à chaque fois que ledit objet déclenchera l'évènement correspondant au type en question, la ou les fonctions d'écouteur impliquées seront appelées.

Seul un objet peut faire apparaître un événement. Nous écouterons donc toujours des événements relatifs à des objets (ici, notre télécopieur).

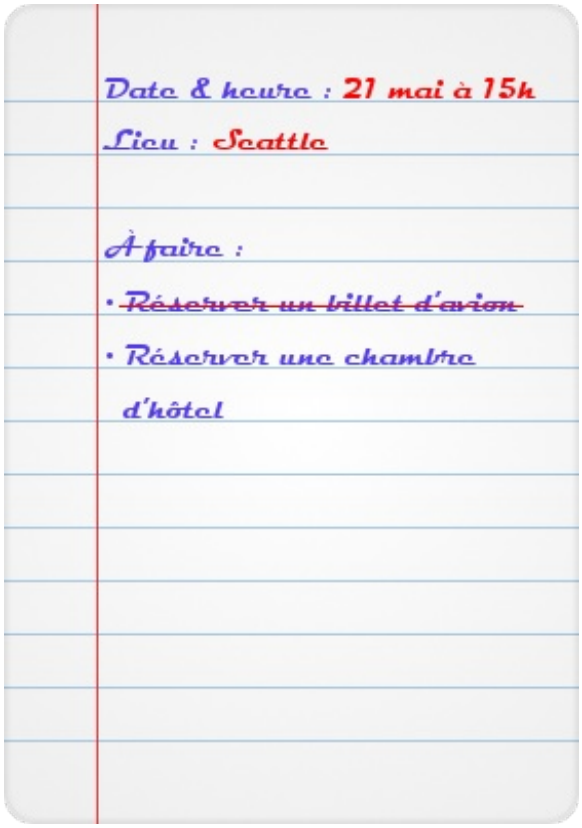
Cependant, le fax n'arrive toujours pas et l'ennui pointe le bout de son nez. Vous levez alors la tête et parcourez votre bureau du regard : vous apercevez un collègue et vous remarquez que ce dernier semble également attendre un fax devant la machine. Toutefois il est peu probable qu'il provienne de Monsieur Gumble. De la même façon, votre voisin n'effectuera pas les mêmes tâches que vous après réception de son fax, sa fonction d'écouteur est donc probablement différente.

Il est possible qu'un événement soit écouté par plusieurs fonctions d'écouteur, mêmes si elles sont différentes. Elles seront toutes appelées lorsque l'événement sera déclenché.

L'évènement

Après la pause déjeuner, vous revenez à votre poste pour poursuivre votre travail de la matinée. Vous n'avez alors pas le temps de vous asseoir, qu'un fax arrive ! Vous regardez celui-ci de plus près, il s'agit bien du fax de monsieur Gumble. Vous le parcourez donc des yeux et voyez que celui-ci regorge d'informations. Vous y trouvez le nom de toutes les personnes qui seront présentes, l'objet de la réunion, mais également la date, l'heure et l'endroit de la réunion. Ça tombe bien, c'est exactement les données qu'il vous manquait.

Vous ressortez alors votre feuille puis commencez les réservations :



Traitement de l'événement « recevoir un fax »

Parmi les différentes informations fournies dans le fax, vous en avez extrait celles qui vous étaient utiles pour pouvoir exécuter les tâches que vous vous étiez fixés. On dit que le fax **représente l'événement** sous la forme d'un objet qui contient différentes données sur ce qui s'est produit.

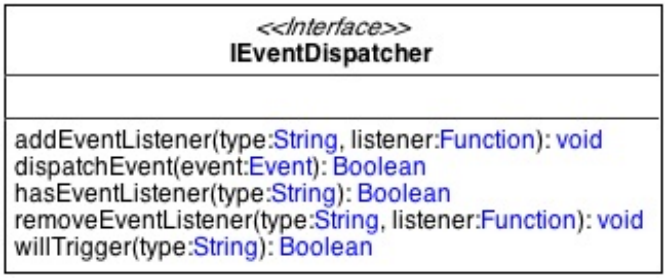
En programmation événementielle et comme tout en Actionscript, les évènements sont donc représentés par des objets, qui renferment également des informations concernant l'évènement en question. De la même façon, il est possible « d'interroger » cette classe, pour pouvoir gérer et traiter correctement ce dernier en fonction de son type et de ses attributs. Ces objets sont tous issus de la classe `Event`, dont nous reparlerons plus loin.

Les écouteurs en POO

Nous allons à présent revenir et nous pencher plus sérieusement sur les écouteurs et leur gestion.
Comme nous l'avons dit plus haut, les écouteurs sont associés aux objets. Ainsi tout objet peut théoriquement utiliser les écouteurs pour répondre aux évènements, à une condition néanmoins : l'objet doit implémenter l'interface `IEventDispatcher`.

L'interface `IEventDispatcher`

L'interface `IEventDispatcher` définit donc le jeu de méthodes servant à la gestion des écouteurs.
Sans plus attendre, voici l'ensemble des définitions de méthodes que propose cette interface :



L'interface IEventDispatcher

Comme vous pouvez le voir, cette interface propose cinq définitions de méthodes pour la gestion des écouteurs. La méthode `addEventListener()` est celle que nous utiliserons le plus, et sa signature complète est la suivante :

Code : Actionscript

```
addEventListener(type:String, listener:Function, useCapture:Boolean
= false, priority:int = 0, useWeakReference:Boolean = false):void
```

Néanmoins dans un premier temps, nous allons laisser de côté les paramètres facultatifs. Ce qui nous donne donc :

Code : Actionscript

```
addEventListener(type:String, listener:Function):void
```

Avant de vous montrer un exemple d'utilisation, nous allons revenir sur la notion d'écouteurs.
Si vous êtes pointilleux, vous aurez alors remarqué la présence du mot anglais « add » dans le nom cette méthode, qui signifie ajout. Ainsi lorsque nous utilisons cette méthode nous ajoutons un nouvel écouteur à notre objet. En effet, il est possible de définir plusieurs écouteurs pour un même objet, et pouvoir ainsi répondre à divers évènements. Grâce à cela, il est donc possible d'être à l'écoute à la fois d'évènements provenant de la souris et d'autres provenant du clavier.

Tout comme il est possible d'ajouter des écouteurs à un objet, il est possible d'en retrancher. C'est donc la méthode `removeEventListener()` qui s'en charge, et voici sa signature complète :

Code : Actionscript

```
removeEventListener(type:String, listener:Function,
useCapture:Boolean = false):void
```

Également, nous retiendrons pour l'instant cette méthode sous sa forme simplifié :

Code : Actionscript

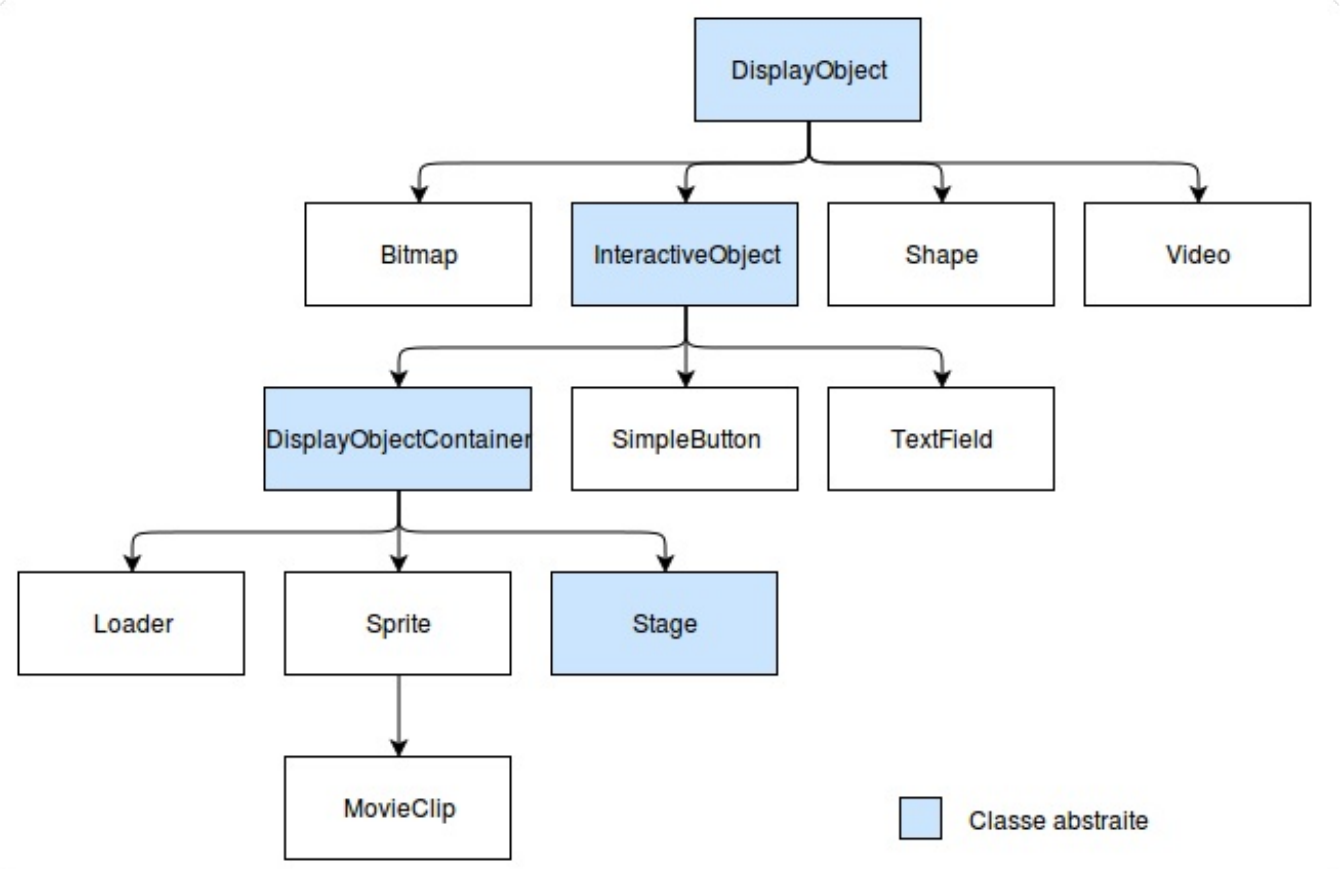
```
removeEventListener(type:String, listener:Function):void
```

Dans un premier temps, nous allons oublier les méthodes `dispatchEvent()` et `willTrigger()`. En revanche, vous pouvez noter la présence de la méthode `hasEventListener()`, qui permet de vérifier si un écouteur est enregistré auprès d'un objet pour un type d'événement spécifique. Cette méthode renvoie donc un booléen.

La classe `EventDispatcher`

Comme nous l'avons dit, c'est l'interface `IEventDispatcher` qui définit le jeu de méthodes servant à la gestion des écouteurs. Néanmoins, c'est à l'intérieur de la classe `EventDispatcher`, que le contenu de l'ensemble de ces méthodes est défini. Bien entendu cette classe implémente l'interface `IEventDispatcher`, et c'est elle qui va réellement servir de classe de base pour toute classe voulant « distribuer » des évènements. Par héritage, cette classe permet donc à tout objet d'utiliser les méthodes de l'interface `IEventDispatcher`, et d'être à l'écoute d'évènements ; on dit alors que l'objet devient une cible d'évènements.

Ainsi toutes les classe filles de `EventDispatcher` héritent donc de ces méthodes liées à la gestion des écouteurs. À l'instar de nombreux objets, nous retrouvons parmi elles la classe `DisplayObject`, que nous avons déjà vue.
Donc si vous vous souvenez bien, l'ensemble des classes d'affichage sont, par héritage, des sous-classes de la classe `EventDispatcher` :



Extrait de l'arbre d'héritage des classes d'affichage

Cela tombe bien, tous les éléments de notre liste d'affichage vont donc pouvoir répondre à divers évènements. Nous verrons que les objets appartenant à la liste d'affichage répondent aux évènements d'une manière assez spécifique : on appelle cela le **flux d'évènements**. Nous aborderons donc cette notion au cours de ce chapitre.

Mise en place d'un écouteur

Introduction aux fonctions de rappel

La théorie

En regardant la méthode `addEventListener()` de plus près, vous vous apercevrez que celle-ci dispose d'un paramètre de type `Function` :

Code : Actionscript

```
addEventListener(type:String, listener:Function):void
```

Dans ce cas, la fonction `listener` est ce qu'on appelle une **fonction de rappel** (ou **callback** en anglais). Il s'agit donc d'une fonction ou d'une méthode qui est passée en paramètre à une autre fonction. Ainsi à l'intérieur de cette dernière, il est possible de faire appel à la *fonction de rappel* à sa guide.

Quel est l'intérêt de cette fonction de rappel ? N'est-il pas plus simple de définir les instructions directement dans la seconde fonction ?

L'intérêt ici est justement de pouvoir se servir d'une fonction, sachant qu'on ne la connaît pas à l'avance. De cette manière, il est alors possible d'exécuter des instructions *dynamiquement* sans avoir besoin de les définir directement. Le plus souvent, les fonctions de rappel sont appelés suite à une condition, comme l'apparition d'un évènement par exemple.
En utilisant ce concept, vous pouvez ainsi créer une classe et laisser à l'utilisateur la liberté d'exécuter les instructions qu'il souhaites, sans qu'il est accès à la classe elle-même. Cela est extrêmement pratique lorsqu'on conçoit son programme par héritage ou lorsqu'on crée une bibliothèque de classes.

Une fonction de rappel est une fonction appelée à l'intérieur d'une autre. Son utilisation et sa signature sont donc définies et figées lors de son appel. Ainsi à la définition de la fonction de rappel, il est nécessaire de respecter les points suivants :

- le nombre et le type des paramètres
- le type de la valeur renvoyée.

Vous pouvez donc tout à fait utiliser ce concept à l'intérieur de vos fonctions et ainsi améliorer la ré-utilisabilité de votre code.

Un exemple

Pour mieux comprendre, voici un exemple basique de fonction utilisant une fonction de rappel suivant une condition :

Code : Actionscript

```
function uneFonction(condition:Boolean, fonction:Function):void
{
    if(condition){
        fonction.call();
    }
}
```

Et voici la définition de la fonction de rappel :

Code : Actionscript

```
function fonctionDeRappel():void
{
    trace("Ceci est une fonction de rappel.");
}
```

Pour utiliser au mieux ce concept, je vous conseille de vous rendre dans la documentation officielle de la classe `Function`. Vous verrez notamment comment vous servir de la méthode `call()` suivant les paramètres définies pour votre fonction.

Si vous souhaitez tester cet exemple, vous pouvez alors exécuter l'instruction suivante, après avoir défini les deux fonctions précédentes :

Code : Actionscript

```
uneFonction(true, fonctionDeRappel);
```

Ce qui donne naturellement à l'exécution du code :

Code : Console

```
Ceci est une fonction de rappel.
```

Créer une fonction d'écouteur

Lorsqu'on parle de gestion d'évènements, les **fonctions de rappel** sont alors plus communément appelées **fonction d'écouteur**, comme décrit plus haut. La gestion de cette fonction est alors du ressort de la superclasse `EventDispatcher`. Lors de la mise en place d'un écouteur, nous avons donc uniquement à nous préoccuper de la définition de la fonction d'écouteur.

Les fonctions d'écouteur doivent donc répondre aux exigences fixées par la classe `EventDispatcher`.
Voici donc la signature type d'une fonction d'écouteur :

Code : Actionscript

```
function uneFonction(event:Event):void
{
    // Instructions à exécutées lors de l'apparition d'un évènement
de type donné
}
```

Le paramètre `event` à renseigner ici est de type `Event`, cependant dans la pratique il s'agira plus généralement d'une sous-classe de cette dernière. La classe utilisée pour ce paramètre dépendra directement du type d'évènement spécifié lors de la mise en place de l'écouteur.

Comme exemple, je vous propose cette fonction d'écouteur qui pourrait servir à la gestion d'évènements liés à la souris :

Code : Actionscript

```
function monEcouteur(event:MouseEvent):void
{
    trace("Vous avez cliqué sur la souris.");
}
```

Comme vous pouvez le voir ici, le paramètre de cette fonction est de type `MouseEvent`, dont nous reparlerons dès le prochain chapitre. Il s'agit donc d'une sous-classe de la classe `Event`, spécifique aux évènements générés par votre souris. Bien évidemment, cette classe aurait été différente s'il s'agissait d'évènements provenant du clavier par exemple.

Gérer les écouteurs d'un objet

Ajouter un écouteur

Comme vous l'aurez compris, pour ajouter un écouteur à un objet, nous aurons besoin de la méthode `addEventListener()`. Parce que mieux vaut deux fois plutôt qu'une, je vous rappelle l'allure de la signature de cette dernière :

Code : Actionscript

```
addEventListener(type:String, listener:Function):void
```

À présent, vous savez donc comment définir la fonction à passer en paramètre de cette méthode. Quant au type d'évènement ici sous forme de `String`, celui-ci peut être renseigné via les constantes définies à l'intérieur de la classe `Event`, ou de l'une de ses sous-classes.

Encore une fois, l'utilisation de ces constantes facilitent grandement la lisibilité et le débogage du code. Par exemple, la chaîne de caractères `"click"` peut être remplacée par l'expression `MouseEvent.CLICK`. De cette manière, le code est plus clair. Par

ailleurs, le compilateur peut alors détecter une éventuelle faute de frappe, ce qui n'est pas le cas pour la chaîne en « brut ».

La constante `MouseEvent.CLICK` permet de faire référence aux évènements générés par le clic de votre souris. En outre, nous sommes capables de terminer la mise en place de notre écouteur.
Voici donc comment enregistrer l'évènement monEcouteur() définie plus haut à l'objet `stage` :

Code : Actionscript

```
stage.addEventListener(MouseEvent.CLICK, monEcouteur);
```

Si vous le désirez, vous pouvez à présent tester le code complet suivant :

Code : Actionscript

```
function monEcouteur(event:MouseEvent):void
{
    trace("Vous avez cliqué sur la scène principale.");
}
stage.addEventListener(MouseEvent.CLICK, monEcouteur);
```

Une fois l'application lancée et si vous cliquez à plusieurs reprise à l'intérieur de votre scène, vous verrez apparaître ceci :

Code : Console

```
Vous avez cliqué sur la scène principale.
Vous avez cliqué sur la scène principale.
Vous avez cliqué sur la scène principale.
```

Dans cet exemple, nous avons enregistré l'écouteur auprès de l'objet `stage`, c'est-à-dire la scène principale. Nous aurions très bien pu le faire pour notre classe `Main` de type `Sprite`. Cependant si vous n'avez rien ajouté à celle-ci, ses dimensions sont nulles et il n'est pas possible d'y cliquer à l'intérieur.

Supprimer un écouteur

À tout moment, il est possible de retirer un écouteur enregistré auprès d'un objet. Pour cela, nous disposons de la méthode `removeEventListener()` définie par l'interface `IEventDispatcher` :

Code : Actionscript

```
removeEventListener(type:String, listener:Function):void
```

Une fois cette méthode appelée, l'objet ne répond alors plus aux évènements en question. Nous n'allons pas épiloguer plus longtemps sur cette méthode qui s'utilise de la même façon que `addEventListener()`.

Voici donc comment retirer l'écouteur ajouté plus haut :

Code : Actionscript

```
stage.removeEventListener(MouseEvent.CLICK, monEcouteur);
```

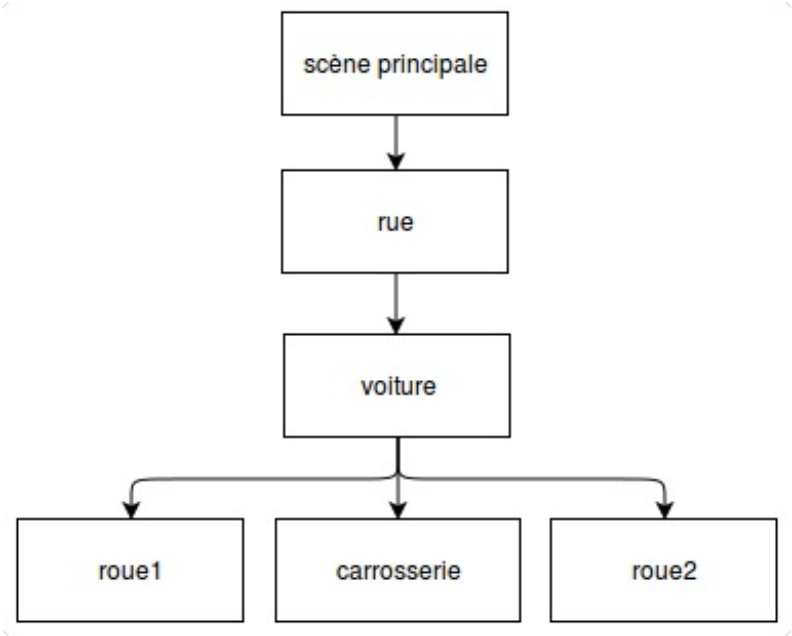
Le flux d'événements

Présentation du concept de flux d'évènements

Introduction

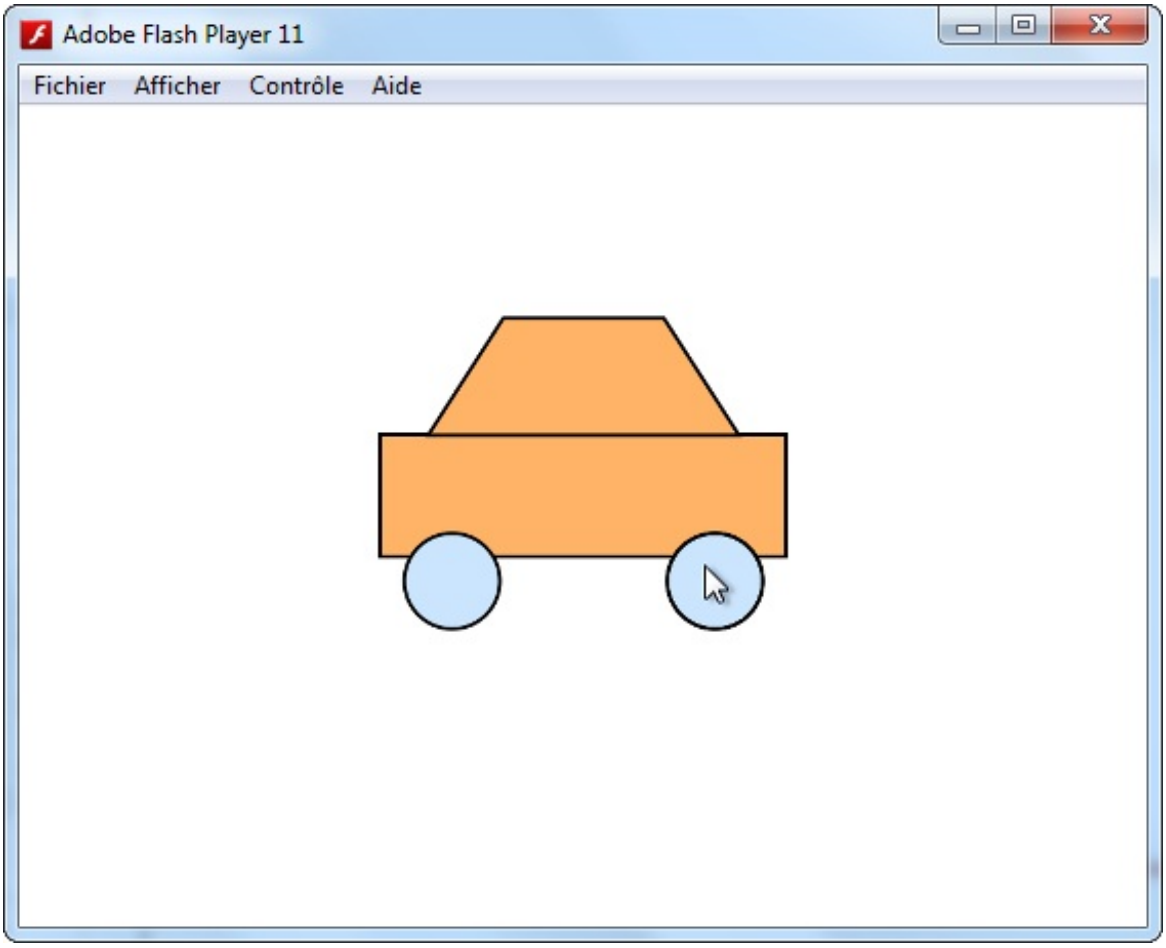
Lorsque survient un évènement, Flash Player distribue celui-ci directement à l'objet cible. Ce dernier peut alors gérer l'évènement comme il le souhaite, grâce à un éventuel écouteur. En revanche, ceci est plus complexe s'il s'agit d'un objet d'affichage. Pour vous en convaincre, nous allons reprendre nos histoires de voitures !

Souvenez-vous de notre voiture organisée de la manière suivante dans la liste d'affichage :



L'arbre d'affichage de la voiture

Il s'agit donc de la hiérarchie à l'intérieur de notre programme. Ceci dit, une fois que vous aurez lancé l'exécution du code, vous obtiendrez la fenêtre suivante :



Les différents objets de la voiture à l'intérieur du Flash Player

Dans l'image précédente, vous voyez que le curseur est positionné sur l'une des deux roues. Au hasard, nous dirons qu'il s'agit de l'objet `roue2`. Si vous cliquez à l'aide de votre souris, un objet `MouseEvent` va donc être généré pour que l'évènement

puisse être traité.
Cependant une petite question se pose alors !

À qui doit être distribué l'évènement généré par la souris ?

Certains diront qu'il semble plus logique de distribuer l'évènement auprès de l'objet `roue2`. En effet, c'est bien l'objet qui réellement la cible de l'évènement, puisque c'est sur lui que le clic est effectué. À première vue, il est normal de faire de l'objet `roue2` la cible de l'évènement.

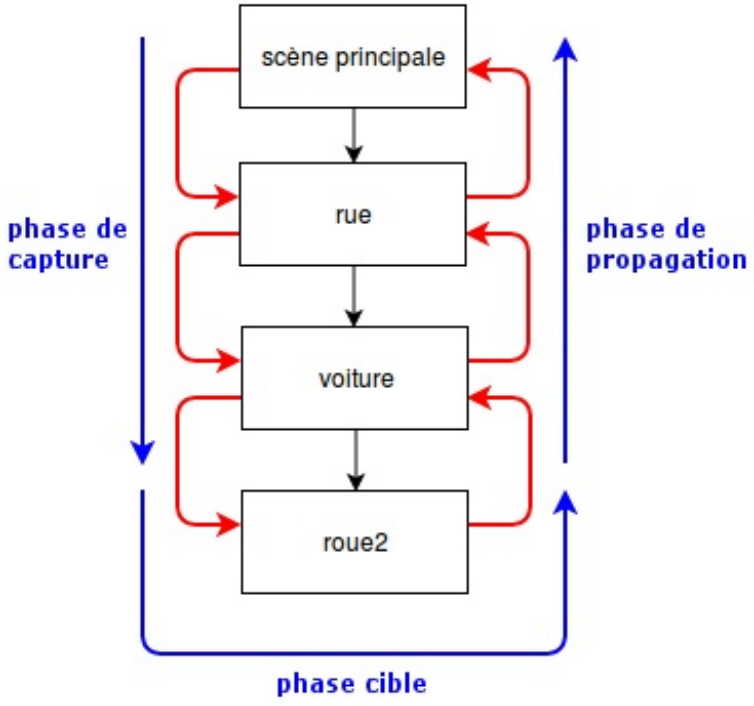
Toutefois en y réfléchissant bien, ceci n'est finalement pas ce qu'il y a de plus pratique. Imaginons que dans notre application nous voulons non pas détecter un clic sur l'une des roues ou même sur la carrosserie, mais sur la voiture elle-même. Comment allons-nous pouvoir gérer cette situation au niveaux des évènements ? Allons nous devoir définir un écouteur pour chacun des objets enfants de notre objet `voiture` ? Heureusement, non ! Le Flash Player intègre ce qu'on appelle le **flux d'évènements**, qui élargit grandement les possibilités de la gestion des évènements.

Qu'est-ce que le flux d'évènements ?

Pour faciliter la gestion des évènements pour les objets d'affichage, le Flash Player utilise une technique qui consiste à acheminer l'objet d'évènement dans toute la hiérarchie de la liste d'affichage, depuis la scène principale jusqu'à l'objet cible. Pour être plus précis, l'objet évènement effectue en réalité un aller-retour sur le chemin décrit précédemment.
Nous pouvons alors distinguer trois phases :

- la **phase de capture** : cette phase comprend la distribution de l'objet évènement depuis la scène principale jusqu'à l'objet parent de la cible,
- la **phase cible** : il s'agit de la gestion de l'évènement par l'objet cible,
- la **phase de propagation** : cette dernière phase est la « remontée » jusqu'à la scène principale, il s'agit donc du chemin inverse à la phase de capture.

Appliqué au cas d'un clic sur l'objet roue2 de notre voiture, le flux d'évènements peut alors être décomposé de la façon suivante :



La diffusion dans le flux d'évènements

Avec cette manière de procéder, il devient possible de gérer l'évènement à tous les étages de la hiérarchie de notre liste d'affichage. Ainsi, nous pouvons alors enregistrer un écouteur pour cet évènement auprès du conteneur `voiture`. Pour mieux comprendre tous les mécanismes du flux d'évènements, nous allons maintenant réaliser quelques manipulations.

Bien comprendre le fonctionnement

Dans le flux d'évènements, l'objet d'évènements est distribué à chaque « nœud » ou objet de la liste d'affichage. Pour bien cerner comment cela fonctionne, nous allons tester quelques petits bouts de code. Nous allons donc reprendre notre voiture.

Commençons par définir nos éléments :

Code : Actionscript

```
// Définition des objets d'affichage
var voiture:Sprite = new Sprite();
voiture.name = "voiture";
var roue2:Sprite = new Sprite();
roue2.name = "roue2";
voiture.addChild(roue2);
addChild(voiture);
```

Pour simplifier l'exemple, nous limiterons le tracé de notre voiture à une unique roue. C'est un peu ridicule, je sais ! Mais cela ne nous empêchera pas de comprendre les mécanismes liés au flux d'évènements.

Voici donc le code proposé pour dessiner cette roue :

Code : Actionscript

```
// Traçage de la roue
roue2.graphics.lineStyle(2, 0x000000);
roue2.graphics.beginFill(0xCCE5FF);
roue2.graphics.drawCircle(0,0,25);
roue2.graphics.endFill();
roue2.x = stage.stageWidth / 2;
roue2.y = stage.stageHeight / 2;
```

En guise de fonction d'écouteur, nous nous contenterons de citer le nom de l'objet d'affichage qui reçoit l'objet évènement. C'est pour cela que nous avons défini la propriété `name` pour chacun de nos objets.

Pour rappel, le nom des variables est arbitraire et n'a de sens qu'à l'intérieur du code. Une fois le code compilé, Flash Player gère directement les variables en mémoire sans utiliser le nom que vous avez défini dans votre code. C'est pourquoi la propriété `name` définie pour toutes les sous-classes de `DisplayObject`, permet facilement de distinguer nos objets d'affichage lors de l'exécution du code.

Voici donc notre fonction d'écouteur :

Code : Actionscript

```
// Définition de la fonction d'écouteur
function monEcouteur(event:MouseEvent):void
{
    trace("Événement en cours de traitement par l'objet " +
event.currentTarget.name);
}
```

Appliquons alors cette fonction d'écouteur à nos deux objets `voiture` et `roue2`, pour les évènements de type `MouseEvent.CLICK` :

Code : Actionscript

```
// Mise en place des écouteurs
voiture.addEventListener(MouseEvent.CLICK, monEcouteur);
roue2.addEventListener(MouseEvent.CLICK, monEcouteur);
```

Je vous propose à présent de lancer l'exécution du code, puis de cliquer sur votre roue à l'aide de votre souris.
Vous verrez alors apparaître ceci dans la console :

Code : Console

```
Événement en cours de traitement par l'objet roue2
Événement en cours de traitement par l'objet voiture
```

Attendez ! Pourquoi l'évènement est-il distribué à l'objet `voiture` après avoir été traité par l'objet `roue2` ?
Et sachant que le flux d'évènements représente un aller-retour dans la liste d'affichage, ne devrait-il pas y avoir deux traitement de la fonction d'écouteur par l'objet `stage` ?

En toute rigueur, si ! Toutefois, lors de la mise en place d'un écouteur avec la méthode `addEventListener()`, nous définissons en réalité la phase durant laquelle l'écouteur est actif. Pour cela, il faut revenir sur la définition de cette fonction :

Code : Actionscript

```
addEventListener(type:String, listener:Function, useCapture:Boolean
= false, priority:int = 0, useWeakReference:Boolean = false):void
```

Dans cette méthode, le troisième paramètre nommé `useCapture`, de type `Boolean`, permet de définir les phases durant lesquelles l'écouteur est actif. Voilà donc comment ce paramètre influe sur le flux d'évènements :

- **`true`** : l'écouteur est actif uniquement pendant la phase de capture,
- **`false`** : l'écouteur est actif pendant les phases cible et de propagation.

Par défaut, ce paramètre vaut **`false`**. Dans le cas de notre objet `voiture`, l'écouteur est donc actif seulement pendant la phase de propagation.

Essayons maintenant de définir l'écouteur de notre objet voiture pendant la phase de capture :

Code : Actionscript

```
voiture.addEventListener(MouseEvent.CLICK, monEcouteur, true);
```

Cette fois-ci, notre `voiture` voit bien l'évènement avant l'objet `roue2` :

Code : Console

```
Événement en cours de traitement par l'objet voiture
Événement en cours de traitement par l'objet roue2
```

Si vous avez bien compris la logique du flux d'évènements, vous aurez alors deviné que l'objet `voiture` peut intervenir à la fois durant la phase de capture *et* la phase de propagation. Cependant, cela nécessite d'enregistrer deux écouteurs différents auprès de l'objet en question.

Même si cela semble superflu, voilà les instructions de définition des écouteurs :

Code : Actionscript

```
voiture.addEventListener(MouseEvent.CLICK, monEcouteur, true);
roue2.addEventListener(MouseEvent.CLICK, monEcouteur);
voiture.addEventListener(MouseEvent.CLICK, monEcouteur);
```

À présent, nous pouvons donc agir à tous les niveaux, ou disons durant toutes les phases du flux d'évènements, comme le montre la console de sortie :

Code : Console

```
Événement en cours de traitement par l'objet voiture
Événement en cours de traitement par l'objet roue2
Événement en cours de traitement par l'objet voiture
```

Enfin avant de passer à la suite, notez que le passage en paramètre de la valeur **true** à la méthode `addEventListener()` de l'objet cible a pour effet de désactiver l'écouteur pendant la phase cible.
Voici un exemple illustrant ces propos :

Code : Actionscript

```
voiture.addEventListener(MouseEvent.CLICK, monEcouteur, true);
roue2.addEventListener(MouseEvent.CLICK, monEcouteur, true);
voiture.addEventListener(MouseEvent.CLICK, monEcouteur);
```

Ce qui donne évidemment dans la console de sortie :

Code : Console

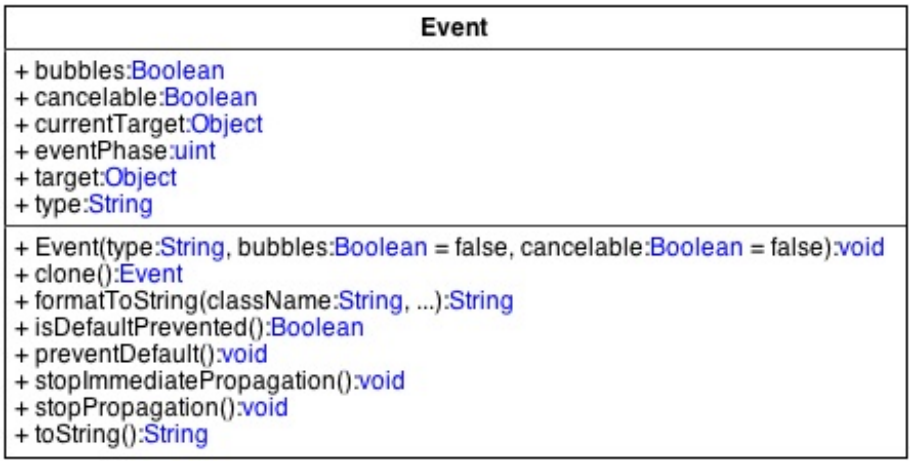
```
Événement en cours de traitement par l'objet voiture
Événement en cours de traitement par l'objet voiture
```

L'objet Event

Présentation de la classe Event

Avant de clôturer ce chapitre, nous allons brièvement parler de la classe `Event`. Nous avons déjà vu que cette dernière est la superclasse à tout objet évènement. Ainsi les sous-classes hériteront de toutes ses propriétés. Ces accesseurs et méthodes serviront principalement à la gestion de l'évènement, nous allons donc rapidement les étudier.

Ainsi voici l'heureuse élue que nous allons étudier :



La classe Event

Bien entendu, nous ne verrons et ne détaillerons pas toutes ces propriétés. Nous nous focaliserons uniquement sur celles qui ont le plus d'intérêt pour nous pour l'instant. Parmi les attributs, certains sont plutôt d'ordre général, et nous retiendrons donc les deux suivants :

- `type` : cet accesseur définit le type de l'évènement sous la forme d'une chaîne de caractères. Il s'agit de cette même chaîne qui est transmise en paramètre à la méthode `addEventListener()`.
- `target` : cet attribut fait référence à l'objet cible de l'évènement.

Voici un petit code d'exemple d'utilisation de ces propriétés :

Code : Actionscript

```
stage.addEventListener(MouseEvent.CLICK, monEcouteur);
function monEcouteur(event:MouseEvent):void
{
```

```
        trace("Événement de type " + event.type);
        trace("Cible de l'événement" + event.target);
    }
```

Ce qui donne :

Code : Console

```
Événement de type click
Cible de l'événement[object Stage]
```

Les propriétés liés au flux d'évènements

Certaines propriétés de la classe `Event` sont en lien direct avec le flux d'évènements. Celles-ci permettent alors de situer et de contrôler l'objet `Event` tout au long du flux d'évènements.
Nous pouvons donc citer les deux attributs suivant :

- `eventPhase` : cette propriété indique la phase en cours dans le flux d'évènements,
- `currentTarget` : cet attribut est une référence à l'objet cible « provisoire », qui peut se situer n'importe où dans la hiérarchie de la liste d'affichage.

Afin de mieux comprendre, nous allons reprendre nos objets `voiture` et `roue2`. Puis nous allons redéfinir la fonction d'écouteur comme ceci, pour faire apparaître les valeurs de ces attributs :

Code : Actionscript

```
// Définition de la fonction d'écouteur
function monEcouteur(event:MouseEvent):void
{
    trace("Événement en cours de traitement par l'objet " +
event.currentTarget.name);
    trace("Phase du flux d'évènements en cours : " +
event.eventPhase);
}
```

Maintenant, enregistrons un écouteur pour chacune des phases du flux d'évènements :

Code : Actionscript

```
// Mise en place des écouteurs
voiture.addEventListener(MouseEvent.CLICK, monEcouteur, true);
voiture.addEventListener(MouseEvent.CLICK, monEcouteur);
roue2.addEventListener(MouseEvent.CLICK, monEcouteur);
```

À présent, il n'y a plus aucun doute sur la manière dont est géré l'objet évènement à travers les différentes phases :

Code : Console

```
Événement en cours de traitement par l'objet voiture
Phase du flux d'évènements en cours : 1
Événement en cours de traitement par l'objet roue2
Phase du flux d'évènements en cours : 2
Événement en cours de traitement par l'objet voiture
Phase du flux d'évènements en cours : 3
```

Comme vous le voyez, les phases sont représentés par un nombre entier allant de 1 à 3. Comme souvent en

Actionscript, l'utilisation de ces valeurs est simplifiée par la classe `EventPhase` et ses constantes respectives `CAPTURING_PHASE`, `AT_TARGET` et `BUBBLING_PHASE`.

Également, la classe `Event` dispose de deux méthodes `stopImmediatePropagation()` et `stopPropagation()` qui servent à stopper la diffusion de l'évènement dans le flux d'évènements, respectivement à partir du nœud actuel ou du nœud suivant. Ces méthodes permettent donc de gérer directement la propagation de l'objet évènement.

Pour visualiser ceci, je vous invite à remplacer la fonction d'écouteur `monEcouteur()` par la suivante :

Code : Actionscript

```
function monEcouteur(event:MouseEvent):void
{
    trace("Événement en cours de traitement par l'objet " +
event.currentTarget.name);
 if(event.eventPhase == EventPhase.AT_TARGET){
  event.stopPropagation();
 }
}
```

La propagation est alors stoppée à la fin de l'exécution de la fonction d'écouteur associée à l'objet `roue2` :

Code : Console

```
Événement en cours de traitement par l'objet voiture
Événement en cours de traitement par l'objet roue2
```

En résumé

- Un **évènement** est généré de manière imprévisible, comme le clic d'une souris par exemple.
- Les méthodes de gestion des écouteurs sont définies par l'interface `IEventDispatcher` et la classe `EventDispatcher`.
- La méthode `addEventListener()` sert à associer une **fonction de rappel** ou **fonction d'écouteur** à un type d'évènements pour un objet donné.
- Le **flux d'évènements** définit le parcours effectué par un évènement dans la hiérarchie de la liste d'affichage, entre la scène principale et l'objet cible.
- Le flux d'évènement est composé de la **phase de capture**, la **phase cible** et la **phase de propagation**.
- Les objets évènement sont représentés par la classe `Event`, ou l'une de ses classes filles.

Intéragir avec l'utilisateur

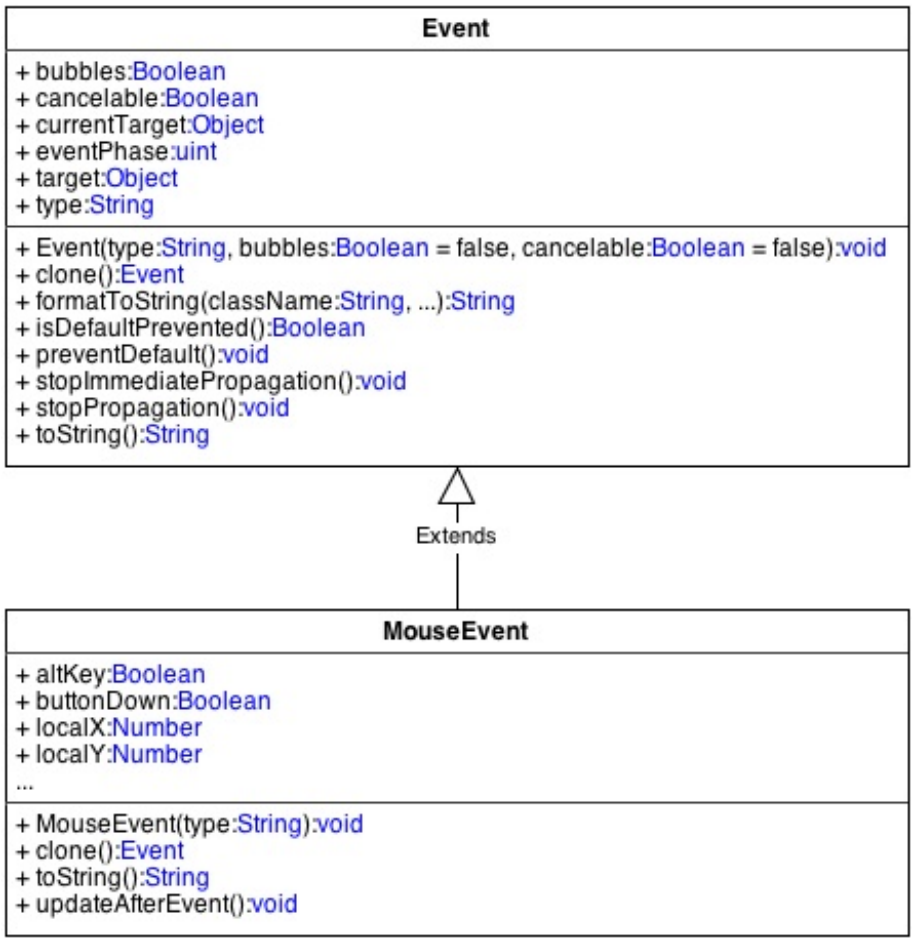
Maintenant que toutes les bases sur la programmation événementielle sont posées, nous allons pouvoir mettre tout ceci en pratique au cours de ce chapitre. Au programme, nous apprendrons comment utiliser et manipuler des données saisies par l'utilisateur. Cela pourra être des données issues de la souris ou du clavier.

La souris

L'objet `MouseEvent`

Introduction

Ici nous allons apprendre à gérer et manipuler des évènements provenant de la souris.
Lorsque qu'un tel évènement se produit, un objet de la classe `MouseEvent` est généré. Comme nous l'avons vu, tous les objets évènements héritent de la classe `Event`, et de ses propriétés :



La classe MouseEvent

Dans le chapitre précédent, j'avais mentionné le fait que les objets renfermaient des informations liés à l'évènement. Ainsi la classe `MouseEvent` dispose principalement de propriétés contenant des informations en lien avec la souris, notamment la position du curseur au moment de la génération de l'évènement. Étant donné que les interactions avec la souris sont étroitement liés à l'interface graphique, cette classe sera évidemment distribuée dans le flux d'évènements.

Nous allons maintenant faire un tour des principales constantes et propriétés de cette classe. Néanmoins, nous ne les verrons pas toutes, et je vous conseille donc de vous rendre sur la page de la classe MouseEvent du guide de référence.

Les différents types d'évènements

Les types d'évènements liés à la souris sont définis sous forme de constantes statiques de la classe `MouseEvent`. On les utilisent ainsi :

Code : Actionscript

```
MouseEvent.NOM_DE_LA_CONSTANTE

// Par exemple :
```

```
        MouseEvent.CLICK
```

Voici ces constantes :

- `CLICK` : cette constante définit un clic avec le bouton gauche de la souris (il faut que le bouton soit successivement enfoncé puis relâché tout en étant au-dessus de la cible à chaque fois).
- `DOUBLE_CLICK` : celle-ci définit un double-clic de la souris. Toutefois la gestion de ce type d'évènements nécessite d'affecter la valeur **`true`** à la propriété `doubleClickEnabled` de l'objet en question.
- `MOUSE_DOWN` : cette constante décrit l'enfoncement du bouton gauche de la souris.
- `MOUSE_UP` : à l'opposé de `MOUSE_DOWN`, `MOUSE_UP` est produit par le relâchement du bouton gauche de la souris.
- `MIDDLE_CLICK` : cette fois-ci, il s'agit d'un clic à l'aide du bouton du milieu de la souris (qui est en général la molette de la souris).
- `MIDDLE_MOUSE_DOWN` : cette constante décrit l'enfoncement du bouton du milieu de la souris.
- `MIDDLE_MOUSE_UP` : comme pour `MOUSE_UP`, `MIDDLE_MOUSE_UP` est produit par le relâchement du bouton du milieu de la souris, à l'opposé de `MIDDLE_MOUSE_DOWN`.
- `RIGHT_CLICK` : pour ce type d'événement, il s'agit d'un clic à l'aide du bouton droit de la souris.
- `RIGHT_MOUSE_DOWN` : cette constante décrit l'enfoncement du bouton droit de la souris.
- `RIGHT_MOUSE_UP` : `RIGHT_MOUSE_UP` est produit par le relâchement du bouton droit de la souris, à l'opposé de `RIGHT_MOUSE_DOWN`.
- `MOUSE_MOVE` : un tel évènement se produit à chaque déplacement de la souris, même d'un seul et unique pixel !
- `MOUSE_WHEEL` : cette constante définit une action de défilement à l'aide de la molette de la souris. La direction de la molette est définie dans l'attribut `delta` de l'objet de l'événement, qui est alors un entier positif ou négatif.
- `ROLL_OVER` : cette constante définit le survol de l'objet courant, ou de l'un de ses enfants, par la souris.
- `ROLL_OUT` : à l'inverse de `MOUSE_OVER`, cet évènement est généré lorsque la souris quitte la « zone de tracé » d'un objet d'affichage, et de l'ensemble de ses enfants.

Grâce à toutes ces constantes que vous pouvez combiner, il vous est maintenant possible de prendre en compte les commande de l'utilisateur avec la souris.

Il existe également les constantes `MOUSE_OVER` et `MOUSE_OUT`, très semblables au couple `ROLL_OVER` / `ROLL_OUT`. La différence se situe au niveau des objets d'affichage enfants de la cible de l'événement : dans le cas de `MOUSE_OVER` et `MOUSE_OUT`, l'événement sera déclenché à chaque fois que la souris quitte la surface d'un enfant pour aller sur un autre, même s'il n'y a pas d'espace vide entre eux. A l'inverse, pour `ROLL_OVER` et `ROLL_OUT`, la cible est ses enfants sont considérés comme étant une unique surface : l'événement n'est déclenché qu'une seule fois. Il est donc fortement conseillé de toujours utiliser `ROLL_OVER` et `ROLL_OUT`.

Imaginons que vous avez créé un bouton constitué d'une image de fond assez grande et d'un champ de texte au milieu, et que vous voulez l'animez au passage de la souris. En utilisant `MOUSE_OVER` et `MOUSE_OUT`, un événement de type `MOUSE_OVER` se déclenchera d'abord lorsque la souris arrivera sur l'image de fond. Mais si la souris passe ensuite sur le champ de texte, un événement de type `MOUSE_OUT` apparaîtra (la souris a quitté la surface d'un enfant du bouton) puis immédiatement après un autre événement de type `MOUSE_OVER` sera déclenché (la souris arrive au-dessus du champ de texte), et vice-versa. Cela n'est pas vraiment optimal, et en plus cela peut provoquer des bugs d'animation pas très agréables sur le bouton lorsque la souris passe dessus.
Si vous utilisez `ROLL_OVER` et `ROLL_OUT`, un événement de type `ROLL_OVER` se déclenchera une seule fois lorsque la souris entrera sur la surface de l'image de fond, même si la souris passe de l'image au champ de texte. Enfin, un unique événement de type `ROLL_OUT` sera déclenché lorsque la souris quittera l'image de fond. En effet, les surfaces de chaque enfant ne sont pas considérées comme indépendantes : elles forment un tout (la surface totale du bouton).

Quelques propriétés utiles

La classe `MouseEvent` possède un certain nombre d'attributs, ou de propriétés. En voici quelques unes qui pourront vous êtes très utiles :

- `stageX` : contient la position horizontale globale où s'est produit l'évènement.
- `stageY` : contient la position verticale globale où s'est produit l'évènement.
- `localX` : contient la position horizontale locale par rapport au conteneur parent où s'est produit l'évènement.
- `localY` : contient la position verticale locale par rapport au conteneur parent où s'est produit l'évènement.
- `buttonDown` : indique si le bouton est enfoncé, principalement utile pour les évènements de type `MOUSE_MOVE`.
- `altKey` : indique si la touche Alt est enfoncée en parallèle de l'événement de la souris.
- `ctrlKey` : indique si la touche Ctrl (ou Cmd pour les utilisateurs de Mac) est enfoncée en parallèle de l'événement de la souris.
- `shiftKey` : indique si la touche Shift est enfoncée en parallèle de l'événement de la souris.
- `delta` (uniquement pour les événements de type `MOUSE_WHEEL`) : contient un entier positif si la molette est défilée

vers le haut, négatif dans le cas contraire.

Les trois propriétés `altKey`, `ctrlKey` et `shiftKey` sont uniquement des indicateurs permettant de déterminer si les touches correspondantes sont enfoncées lors de la génération de l'évènement. Des actions sur ces boutons n'engendrent en aucun cas l'émission d'un objet `MouseEvent`.

Exemple: bouton simple

Pour illustrer ce que nous venons de voir, créons notre premier bouton ensemble !
Voici le code à écrire pour le bouton :

Code : Actionscript

```
// Bouton
var bouton:Sprite = new Sprite();
// Autorisation du double-clic
bouton.doubleClickEnabled = true;
// Curseur de bouton (généralement en forme de main)
bouton.buttonMode = true;
bouton.useHandCursor = true;
// Les enfants ne sont pas gérés par la souris afin qu'ils
n'interfèrent pas avec le curseur de la souris
// (sinon le champ de texte pourrait annuler le curseur de bouton)
bouton.mouseChildren = false;
addChild(bouton);

// Texte
var texte:TextField = new TextField();
// Pour éviter que le texte soit sélectionné
texte.selectable = false;
// Taille automatique
texte.autoSize = TextFieldAutoSize.LEFT;
// Format du texte
texte.defaultTextFormat = new TextFormat('Arial', 32, 0xffffff);
// Contenu
texte.text = "Je suis un bouton";
// Ajout du texte dans le bouton
bouton.addChild(texte);

// Fond bleu
var fond:Shape = new Shape();
// Ligne
fond.graphics.lineStyle(2, 0x9999ff, 1, true);
// Remplissage
fond.graphics.beginFill(0x0000ff);
// Dessin d'un rectangle à coins arrondis
fond.graphics.drawRoundRect(0, 0, texte.width, texte.height, 12);
// Ajout du fond bleu dans le bouton, en dessous du texte (cad, à
l'index 0)
bouton.addChildAt(fond, 0);

// Alignement du bouton au centre de la scène
bouton.x = (stage.stageWidth - bouton.width) / 2;
bouton.y = (stage.stageHeight - bouton.height) / 2;
```

Nous créons en premier lieu un *sprite* qui va représenter le bouton. On active le double clic à l'aide de l'attribut `doubleClickEnabled`, puis on active le curseur de bouton (généralement en forme de main) en mettant les attributs `buttonMode` et `useHandCursor` à **`true`**.

Pourquoi désactiver la gestion de la souris sur les enfants du bouton ?

Si nous ne le faisons pas, le champ de texte aura le fâcheux comportement de mal faire fonctionner le curseur, qui sera remis en

pointeur par défaut (au lieu du curseur de bouton en forme de main). Ainsi, par sécurité, il vaut mieux mettre l'attribut `mouseChildren` à **`false`** pour les enfants soient plus directement gérés par la souris. Toutefois, ils comptent quand-même dans la surface totale du bouton.

Puis nous créons le champ de texte qui affichera l'étiquette du bouton, suivie d'un dessin de rectangle arrondi bleu pour embellir tout ceci.

Ensuite, écrivons les fonctions d'écouteur qui nous permettront de tester différents événements de la souris :

Code : Actionscript

```
// Fonctions d'écouteurs
function onOver(e:MouseEvent):void
{
    trace("La souris est entrée au-dessus du bouton.");
}
function onOut(e:MouseEvent):void
{
    trace("La souris n'est plus au-dessus du bouton.");
}
function onDown(e:MouseEvent):void
{
    trace("Le bouton gauche de la souris est enfoncé au-dessus du
bouton.");
}
function onUp(e:MouseEvent):void
{
    trace("Le bouton gauche de la souris est relâché au-dessus du
bouton.");
}
function onClick(e:MouseEvent):void
{
    trace("Le bouton a été cliqué.");
}
function onDoubleClick(e:MouseEvent):void
{
    trace("Le bouton a été double-cliqué.");
}
function onMiddleClick(e:MouseEvent):void
{
    trace("Le bouton a été cliqué milieu.");
}
function onRightClick(e:MouseEvent):void
{
    trace("Le bouton a été cliqué droit.");
}
function onMove(e:MouseEvent):void
{
    trace("La souris s'est déplacée au-dessus du bouton.");
}
```

Enfin, nous écoutons les différents événements de la souris sur le bouton, qui en sera alors la **cible** :

Code : Actionscript

```
// Evénements de la souris
bouton.addEventListener(MouseEvent.ROLL_OVER, onOver);
bouton.addEventListener(MouseEvent.ROLL_OUT, onOut);
bouton.addEventListener(MouseEvent.MOUSE_DOWN, onDown);
bouton.addEventListener(MouseEvent.MOUSE_UP, onUp);
bouton.addEventListener(MouseEvent.CLICK, onClick);
bouton.addEventListener(MouseEvent.DOUBLE_CLICK, onDoubleClick);
bouton.addEventListener(MouseEvent.MIDDLE_CLICK, onMiddleClick);
bouton.addEventListener(MouseEvent.RIGHT_CLICK, onRightClick);
bouton.addEventListener(MouseEvent.MOUSE_MOVE, onMove);
```

Nous avons quelque chose comme ceci à l'écran :

Notre bouton bleu !

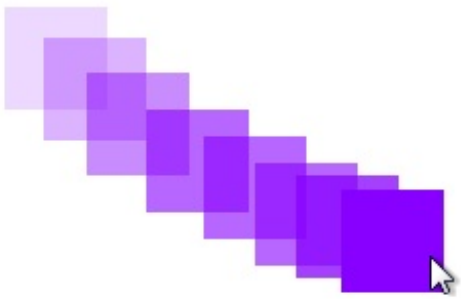
Si vous jouez un peu avec, vous obtiendrez des informations dans la console sur les différents événements qui se déclenchent :

Code : Console

```
La souris est entrée au-dessus du bouton.
La souris s'est déplacée au-dessus du bouton.
La souris s'est déplacée au-dessus du bouton.
La souris s'est déplacée au-dessus du bouton.
Le bouton gauche de la souris est enfoncé au-dessus du bouton.
Le bouton gauche de la souris est relâché au-dessus du bouton.
Le bouton a été cliqué.
Le bouton a été cliqué milieu.
Le bouton a été cliqué droit.
La souris s'est déplacée au-dessus du bouton.
La souris s'est déplacée au-dessus du bouton.
La souris s'est déplacée au-dessus du bouton.
La souris n'est plus au-dessus du bouton.
```

La technique du « glisser-déposer »

Le **glisser-déposer** (ou **drag and drop** en anglais) est une méthode qui consiste à déplacer un élément graphique d'un endroit à un autre. Il s'agit du même principe que lorsque que vous déplacez une icône sur votre bureau. Pour cela, vous effectuez un clic que vous maintenez le temps du déplacement. Puis le déplacement s'arrête lorsque vous relâchez le bouton de la souris.



Le « glisser-déposer »

En Actionscript, ce concept est beaucoup plus poussé et puissant, puisqu'il est possible de contrôler à souhaits ce fameux « glisser-déposer ». Une manière de réaliser ceci, serait de combiner intelligemment les trois types d'évènements `MOUSE_DOWN`, `MOUSE_MOVE` et `MOUVE_UP`. Mais l'Actionscript facilite davantage la mise en place de cette technique. C'est pourquoi la classe `Sprite` dispose des deux méthodes `startDrag()` et `stopDrag()`, permettant respectivement d'activer ou de désactiver le glisser-déposer sur cet objet.

Ici, nous allons utiliser un exemple qui sera beaucoup plus parlant. Commençons par dessiner un carré violet dans notre scène :

Code : Actionscript

```
var objet:Sprite = new Sprite();
addChild(objet);
objet.graphics.beginFill(0x8800FF);
objet.graphics.drawRect(0, 0, 50, 50);
```

En utilisant ces fameuses méthodes `startDrag()` et `stopDrag()`, il n'est maintenant plus nécessaire de se préoccuper des déplacements de la souris. Seuls le démarrage et l'arrêt du glisser-déposer doivent être contrôlés. Nous pouvons alors très facilement mettre ceci en place grâce aux deux écouteurs suivants :

Code : Actionscript

```
objet.addEventListener(MouseEvent.MOUSE_DOWN, glisser);
function glisser(event:MouseEvent):void
{
    objet.startDrag();
}
objet.addEventListener(MouseEvent.MOUSE_UP, deposer);
function deposer(e:MouseEvent):void
{
    objet.stopDrag();
}
```

À l'intérieur du Flash Player, il est maintenant possible de déplacer notre objet à volonté !

Exercice : Créer et animer un viseur

Dessiner le viseur

Dans cet exercice, nous allons voir comment dessiner et animer un viseur, tel que nous pourrions en trouver dans un jeu de tir. Nous commencerons donc par tracer l'élément qui vous servira de viseur.

Voici le viseur que je vous propose de dessiner :


Le viseur

Créons donc un objet `viseur` de type `Shape`, et traçons les différents éléments qui compose celui-ci :

Code : Actionscript

```
var viseur:Shape = new Shape();
addChild(viseur);
viseur.graphics.lineStyle(2, 0x000000);
viseur.graphics.drawCircle(0, 0, 20);
viseur.graphics.lineStyle(1, 0x550000);
viseur.graphics.drawCircle(0, 0, 16);
viseur.graphics.moveTo(-25, 0);
viseur.graphics.lineTo(25, 0);
viseur.graphics.moveTo(0, -25);
viseur.graphics.lineTo(0, 25);
viseur.x = stage.stageWidth / 2;
viseur.y = stage.stageHeight / 2;
```

Comme vous pouvez le voir, il n'y a rien de bien compliqué dans ce code. Nous avons uniquement tracé deux cercles concentriques et deux segments perpendiculaires.

En réalité pour cet exercice, nous n'allons pas réellement remplacé le curseur. Nous allons nous contenter de « suivre » la position de la souris, et nous masquerons la souris elle-même. Cela donnera alors l'illusion que le curseur est un viseur.

Pour masquer la souris, utilisez l'instruction suivante :

Code : Actionscript

```
Mouse.hide();
```

La fonction `hide()` est une méthode statique de la classe `Mouse`. N'oubliez donc pas d'importer cette classe dans votre projet. Pour faire réapparaître votre curseur, vous pouvez utiliser la méthode statique inverse : `show()`.

Gérer les évènements

À présent, occupons-nous de la gestion des évènements !
Nous allons donc faire ça en deux temps. Premièrement, nous allons faire en sorte que le viseur suive le curseur en mettant à jour sa position. Ensuite, nous ajouterons un second écouteur pour simuler un tir . Bien sûr tout cela sera basique, mais l'idée sera là.

Pour mettre à jour la position du viseur, nous allons utiliser les évènements de type `MOUSE_MOVE`. Les propriétés `stageX` et `stageY` de la classe `MouseEvent` nous serviront à replacer le viseur.

Voilà donc la mise en place de notre écouteur :

Code : Actionscript

```
stage.addEventListener(MouseEvent.MOUSE_MOVE, repositionnerViseur);
function repositionnerViseur(event:MouseEvent):void
{
    viseur.x = event.stageX;
    viseur.y = event.stageY;
}
```

Nous allons maintenant nous concentrer sur la gestion des évènements de type « tirs ». Je vous propose pour cela, d'utiliser le clic de la souris, ou plutôt l'enfoncement du bouton de gauche. Il s'agit bien évidemment de l'évènement `MOUSE_DOWN`.

À l'intérieur de la fonction d'écouteur associée, nous créerons un petit rond rouge, grossièrement semblable à une tâche de sang. Puis nous la positionnerons au niveau du viseur. Rien de bien complexe ici, voici le code correspondant :

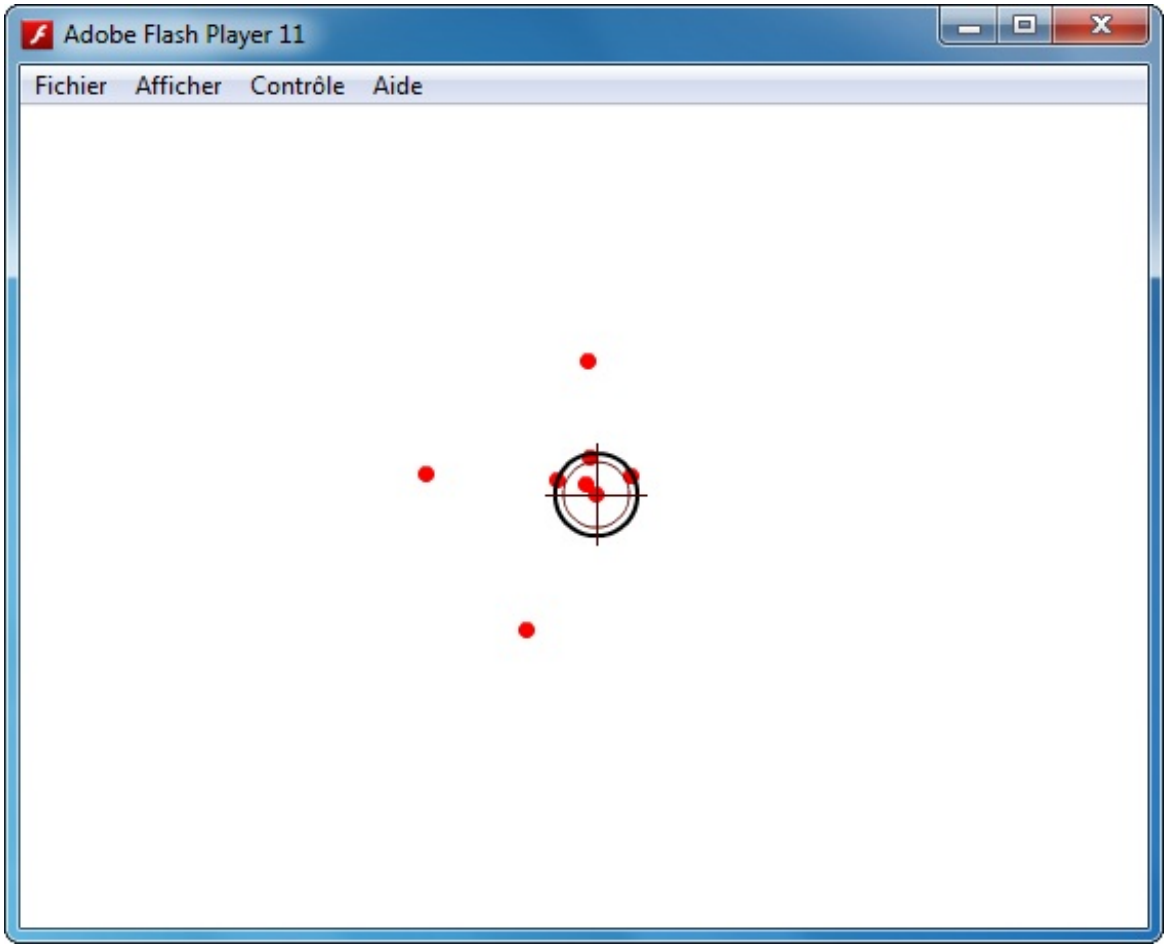
Code : Actionscript

```
stage.addEventListener(MouseEvent.MOUSE_DOWN, tirer);
function tirer(event:MouseEvent):void
{
    var impact:Shape = new Shape();
    impact.graphics.beginFill(0xFF0000);
    impact.graphics.drawCircle(0, 0, 4);
    impact.x = event.stageX;
    impact.y = event.stageY;
    addChildAt(impact, 0);
}
```

Ici, nous avons utilisé un évènement de type `MOUSE_DOWN`, mais nous pourrions très bien pu le faire à l'aide des évènements `MOUSE_UP` ou `CLICK`. Lorsque vous programmer votre interface, c'est à vous de choisir le type d'évènements qui correspond le mieux à ce que vous souhaitez faire.

Le rendu

Pour admirer le magnifique rendu de cette application, je ne peux que vous inviter à exécuter le code, et à tirer dans tous les sens !!!



Les prémices d'un jeu de tir

Curseurs personnalisés

Il est possible de remplacer le curseur de la souris par une image ou une animation de notre choix, ce qui présente des avantages par rapport à la technique que nous venons de voir pour le viseur : le curseur sera un curseur natif, c'est-à-dire qu'il sera directement géré par le système d'exploitation de l'utilisateur. Cela implique de meilleures performances : moins de ressources seront consommé pour afficher le curseur et même si votre application est victime de ralentissement, votre curseur personnalisé n'en sera pas affecté, ce qui améliore grandement l'expérience utilisateur. De plus, le curseur ne sera pas coupé aux bords de la zone d'affichage de votre application : il pourra dépasser en dehors de la fenêtre (par exemple, s'il se situe en bas de votre application, il continuera d'être affiché, même en dehors de la scène). Il est donc très fortement conseillé d'utiliser les curseurs natifs personnalisés si vous voulez remplacer le curseur de la souris : c'est la meilleure méthode.

Il est possible que le système sur lequel votre application est lancée ne supporte pas les curseurs natifs (c'est par exemple le cas pour les smartphones) : vous pouvez le savoir avec l'attribut statique `supportsNativeCursor` de la classe `Mouse`.

Ainsi, nous n'utiliserons de curseur natif que si cette fonctionnalité est supportée :

Code : Actionscript

```
if(Mouse.supportsNativeCursor)
{
    // Création du curseur natif personnalisé ici
}
```

Un curseur natif ne peut pas dépasser une taille de 32 pixels sur 32 pixels.

Préparer son curseur

Il faut savoir qu'un curseur personnalisé ne peut être qu'une image bitmap (ou une série d'images dans le cas des curseurs animés). Ce qui signifie qu'il nous est impossible d'utiliser directement un objet d'affichage comme le dessin (de la classe `Shape`) que nous avons créé pour afficher notre viseur. Toutefois, il existe un moyen de prendre une capture de n'importe quel objet d'affichage pour obtenir une image bitmap ! 

 Il est bien entendu également possible d'utiliser directement des images que vous aurez incorporées au préalable. 

Reprenons le code que nous avons utilisé pour créer notre viseur sans l'ajouter à la scène :

Code : Actionscript

```
var viseur:Shape = new Shape();
viseur.graphics.lineStyle(2, 0x000000);
viseur.graphics.drawCircle(0, 0, 20);
viseur.graphics.lineStyle(1, 0x550000);
viseur.graphics.drawCircle(0, 0, 16);
viseur.graphics.moveTo(-25, 0);
viseur.graphics.lineTo(25, 0);
viseur.graphics.moveTo(0, -25);
viseur.graphics.lineTo(0, 25);
```

Nous ne pouvons pas utiliser ce dessin pour en faire un curseur natif, mais nous pouvons en prendre une capture (comme vous prendriez une capture d'écran). Pour cela, nous allons d'abord créer une image vide avec la classe `BitmapData` et utiliser sa méthode `draw()` (qui signifie 'dessiner'). Cette méthode prend en paramètre n'importe quel objet qui implémente l'interface `IBitmapDrawable`. Comme nous pouvons le voir dans la documentation, il s'agit de tous les objets ayant pour classe (ou super-classe) `BitmapData` ou `DisplayObject`.

Pour créer l'image bitmap, il nous faut les bornes d'affichage de notre viseur : en effet, notre dessin est centré par rapport à son origine, donc si nous en faisons une capture sans en tenir compte, il sera coupé à partir de son origine (donc nous n'obtiendrons qu'un quart du viseur). Nous calculons les bornes d'affichage de notre dessin grâce à la méthode `getBounds()` de la classe `DisplayObject`, qui prend en paramètre un objet d'affichage servant de référence. Ici, nous passons donc le viseur lui-même pour savoir de combien il dépasse par rapport à sa propre origine :

Code : Actionscript

```
// Bornes d'affichage de l'objet
var bornes:Rectangle = viseur.getBounds(viseur);
trace("bornes:" + bornes);
```

Si vous testez votre code tout de suite, vous obtiendrez ceci dans la console :

Code : Console

```
bornes:(x=-25.5, y=-25.5, w=51, h=51)
```

Nous pouvons alors remarquer que notre dessin dépasse de 25 pixels et demi par rapport à son origine, verticalement et horizontalement. Sa taille totale est de 51 pixels sur 51 pixels.

Créons notre image vide :

Code : Actionscript

```
// Image vide de bonne taille
var capture:BitmapData = new BitmapData(32, 32, true, 0);
```

Ensuite, pour prendre en compte le fait que notre dessin dépasse de son origine, nous allons "déplacer" la capture pour compenser, à l'aide d'une matrice de transformation :

Code : Actionscript

```
// Déplacement de la capture en fonction des bornes d'affichage
var transformation:Matrix = new Matrix();
transformation.translate( -bornes.x, -bornes.y);
```

Comme un curseur natif ne peut dépasser une taille de 32 pixels de côté, nous allons réduire la taille du viseur :

Code : Actionscript

```
// La taille maximale d'un curseur natif est de 32 pixels
transformation.scale(32 / bornes.width, 32 / bornes.height);
```

Enfin, il ne nous reste plus qu'à prendre la capture de notre viseur :

Code : Actionscript

```
// Capture
capture.draw(viseur, transformation);
```

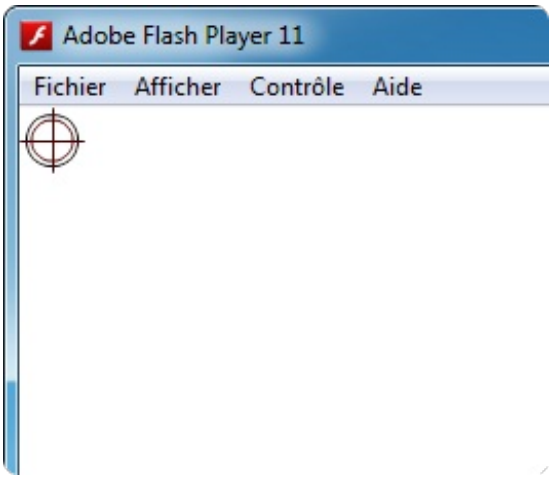
Désormais, notre image contient une capture bitmap de notre viseur, utilisable pour créer un curseur personnalisé !

Pour vérifier que nous ne nous sommes pas trompés, affichons la capture avec un objet de la classe `Bitmap` :

Code : Actionscript

```
// Affichage de la capture pour vérification
var apercu:Bitmap = new Bitmap(capture);
addChild(apercu);
```

Nous obtenons alors ceci :



L'aperçu de la capture

Pour bien comprendre qu'il s'agit d'une capture bitmap, agrandissons l'aperçu quatre fois :

Code : Actionscript

```
apercu.scaleX = 4;
apercu.scaleY = apercu.scaleX;
```

Ce qui nous donne :

Il s'agit bien d'une image bitmap

Les gros pixels nous confirment que nous avons là une image bitmap et non plus un dessin vectoriel.

Créer le curseur natif

Chaque curseur natif est créé à l'aide d'un objet de la classe `MouseCursorData` contenant les images et autres informations à propos du curseur. Commençons donc par en créer une instance :

Code : Actionscript

```
// Données du curseur natif personnalisé
var curseur:MouseCursorData = new MouseCursorData();
```

Ensuite, il nous faut un vecteur contenant une ou plusieurs images (ici nous lui insérons la capture du viseur) :

Code : Actionscript

```
// Images du curseur (s'il y en a plusieurs, le curseur sera animé)
var images:Vector.<BitmapData> = new Vector.<BitmapData>();
images.push(capture);
curseur.data = images;
```

Puis, nous renseignons le **point actif** du curseur, c'est-à-dire la position qui représente le point où l'utilisateur va cliquer. Dans notre cas, nous voulons que le point actif soit le centre du viseur :

Code : Actionscript

```
// Point actif du curseur -> centre de notre viseur
curseur.hotSpot = new Point(capture.width / 2, capture.height / 2);
```

Enfin, il faut enregistrer notre curseur natif en lui donnant un nom, grâce à la méthode statique `registerCursor` de la classe `Mouse` :

Code : Actionscript

```
// Enregistrement du cuseur natif personnalisé
Mouse.registerCursor("viseur", curseur);
```

Notre curseur est fin prêt à être utilisé !

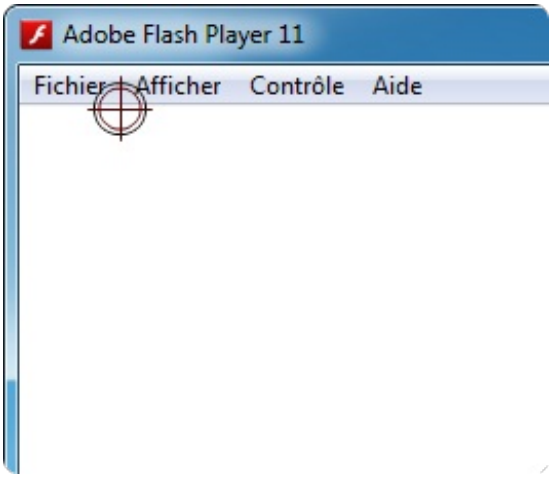
Activer un curseur natif

Pour activer notre nouveau curseur natif, il faut affecter à l'attribut statique `cursor` de la classe `Mouse` le nom de notre curseur :

Code : Actionscript

```
// Activons notre cuseur
Mouse.cursor = "viseur";
```

Et voilà que notre curseur de souris s'est transformé en viseur :



Notre curseur natif personnalisé

Comme vous pouvez le remarquer, le curseur n'est plus coupé par les bords de notre application, et il est très fluide quelque soit la fréquence de rafraîchissement de l'application.

Voici le code complet permettant de créer ce curseur natif :

Code : Actionscript

```
if (Mouse.supportsNativeCursor)
{
    // Dessin
    var viseur:Shape = new Shape();
    viseur.graphics.lineStyle(2, 0x000000);
    viseur.graphics.drawCircle(0, 0, 20);
    viseur.graphics.lineStyle(1, 0x550000);
    viseur.graphics.drawCircle(0, 0, 16);
    viseur.graphics.moveTo(-25, 0);
    viseur.graphics.lineTo(25, 0);
    viseur.graphics.moveTo(0, -25);
    viseur.graphics.lineTo(0, 25);

    // Bornes d'affichage de l'objet
    var bornes:Rectangle = viseur.getBounds(viseur);
    trace("bornes:" + bornes);

    // Image vide de bonne taille
    var capture:BitmapData = new BitmapData(32, 32, true, 0);
    // Déplacement de la capture en fonction des bornes d'affichage
    var transformation:Matrix = new Matrix();
    transformation.translate( -bornes.x, -bornes.y);
    // La taille maximale d'un curseur natif est de 32 pixels
```

```
            transformation.scale(32 / bornes.width, 32 / bornes.height);
            // Capture
            capture.draw(viseur, transformation);

            // Affichage de la capture pour vérification
            /*var apercu:Bitmap = new Bitmap(capture);
        addChild(apercu);
        apercu.scaleX = 4;
        apercu.scaleY = apercu.scaleX;*/


            // Données du curseur natif personnalisé
            var curseur:MouseCursorData = new MouseCursorData();
            // Images du curseur (s'il y en a plusieurs, le curseur sera
        animé)
            var images:Vector.<BitmapData> = new Vector.<BitmapData>();
            images.push(capture);
            curseur.data = images;
            // Point actif du curseur -> centre de notre viseur
            curseur.hotSpot = new Point(capture.width / 2, capture.height /
        2);
            // Enregistrement du cuseur natif personnalisé
            Mouse.registerCursor("viseur", curseur);

            // Activons notre cuseur
            Mouse.cursor = "viseur";

        }
```

Désactiver le curseur natif personnalisé

Si vous n'avez plus besoin de votre curseur natif en forme de viseur, vous pouvez très simplement revenir au curseur par défaut du système d'exploitation de l'utilisateur à l'aide de l'instruction suivante :

Code : Actionscript

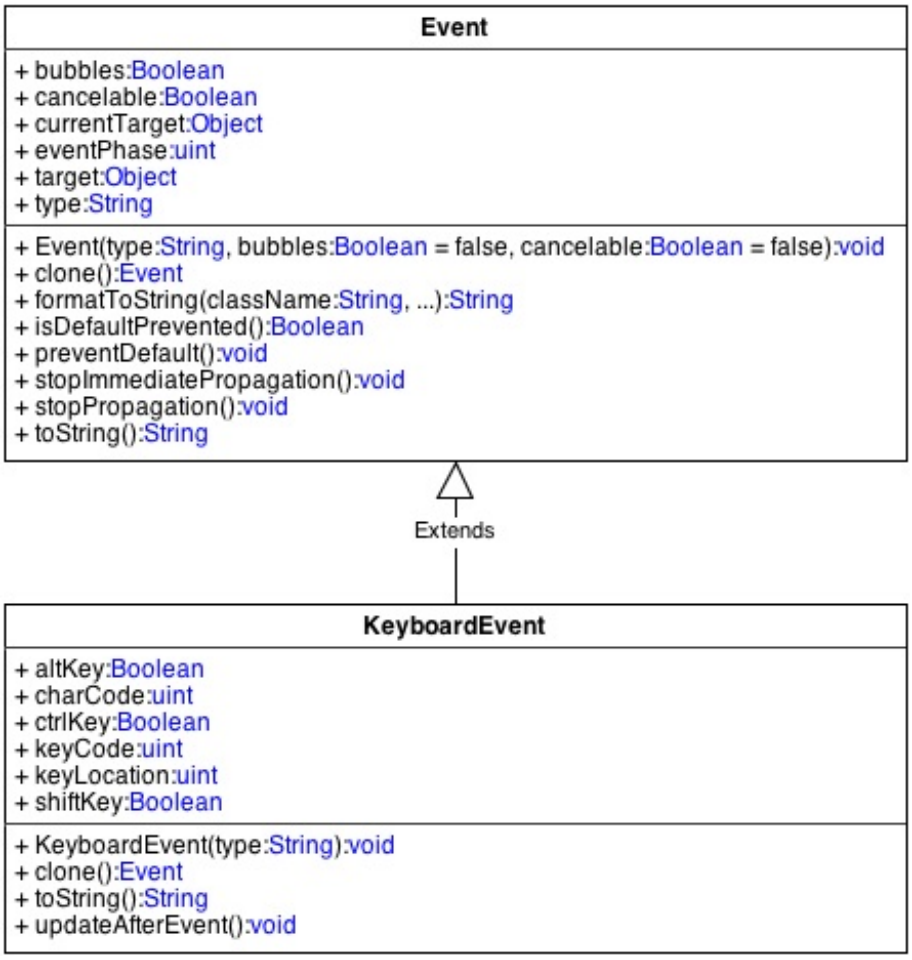
```
Mouse.cursor = MouseCursor.AUTO;
```

Le clavier

L'objet `KeyboardEvent`

Tout comme ceux qui proviennent de la souris, les évènements issus du clavier génèrent leur propre classe, à savoir `KeyboardEvent`. Celle-ci hérite évidemment de la classe `Event`, en y ajoutant ses propres propriétés.

Voici un petit schéma UML qui permet de résumer un peu toutes les propriétés ou méthodes fournies par ces deux classes :



La classe KeyboardEvent

Contrairement à la souris, les types d'évènements produits par le clavier sont beaucoup moins nombreux, puisqu'ils ne sont que deux !
Voici donc les constantes de la classe `KeyboardEvent` qui les définissent :

- `KEY_DOWN` : cette constante décrit l'enfoncement de n'importe quelle touche du clavier.
- `KEY_UP` : celle-ci représente le relâchement de n'importe quelle touche du clavier.

Même s'il est possible de trouver un grand nombre d'applications nécessitant la combinaison de ces deux types d'évènements, vous n'en utiliserez qu'un la majorité du temps. Et comme cela semble souvent plus naturel, il s'agira de l'évènement `KEY_DOWN`.

Les attributs, ou propriétés, associées à la classe `KeyboardEvent` ne sont également pas très nombreuses, et les voici :

- `charCode` : cet attribut représente la lettre, ou plutôt le caractère, de la touche concernée par l'évènement. Ce caractère est définit par un code numérique.
- `keyCode` : cette propriété contient le code de la touche enfoncée ou relâchée.
- `keyLocation` : indique l'emplacement de la touche sur le clavier.
- `altKey` : indique si la touche Alt est enfoncée en parallèle de l'événement de la souris.
- `ctrlKey` : indique si la touche Ctrl (ou Cmd pour les utilisateurs de Mac) est enfoncée en parallèle de l'événement de la souris.
- `shiftKey` : indique si la touche Shift est enfoncée en parallèle de l'événement de la souris.

Pour savoir sur quelle touche l'utilisateur a appuyé, nous pouvons utiliser la classe `Keyboard` qui contient le code de chaque touche du clavier. Par exemple, pour effectuer une action sur l'utilisateur appuie sur la touche A, nous procéderons ainsi :

Code : Actionscript

```
if(event.keyCode == Keyboard.A)
{
    trace("Appui sur la touche [A]");
}
```

Il est important de bien distinguer les deux propriétés `charCode` et `keyCode`. La première désigne le code de la touche dans le jeu de caractères actuel. Chaque caractère est associé à une valeur numérique : on parle alors de **codage de caractères** ou encore **encodage**. Par défaut, il s'agit de l'**UTF-8**, qui prend en charge le célèbre code **ASCII**. En revanche, `keyCode` fait référence à la touche en elle-même indépendamment du codage utilisé. Ainsi les lettres « a » et « A » sont associées à deux codes différent en UTF-8, mais la touche reste la même, et donc `keyCode` aussi. Inversement la touche 1 du pavé numérique est différente du 1 du clavier central, pourtant le caractère est strictement le même.

Enfin avant de basculer sur des exemples, nous allons revenir sur la propriété `keyLocation`. Si vous regardez votre clavier de plus prêt, vous verrez alors que celui-ci est composé de deux touches Ctrl distincts, ou encore de deux touches Shift différentes. Et aussi surprenant que cela puisse paraître, chaque couple possèdent exactement les mêmes attributs `charCode` et `keyCode`.

Mais comment les différencier alors ?

C'est là qu'entre en scène cette fameuse propriété `keyLocation`. Sous forme de nombre, cette propriété permet de définir l'emplacement de la touche sur le clavier. Pour cela, la classe `KeyLocation` propose un jeu de constantes relativement explicite. Jugez plutôt : `STANDARD`, `LEFT`, `RIGHT` et `NUM_PAD`.

Exercice : gérer l'affichage de l'animation

C'est l'heure de passer un peu à la pratique à travers un petit exemple, simple certes, mais très intéressant.
L'objectif est donc de basculer l'animation en mode plein écran avec la combinaison Shift + Entrée (et uniquement à l'aide la touche Shift de gauche). Puis pour revenir à l'affichage normal, un simple appui sur la touche Échap devrait suffire.

Pour réaliser le basculement d'un mode d'affichage à un autre, nous aurons besoin d'utiliser la propriété `displayState` définit par la classe `Stage`. Cette propriété peut alors prendre les valeurs des constantes `NORMAL`, `FULL_SCREEN` et `FULL_SCREEN_INTERACTIVE` de la classe `StageDisplayState`.

Le mode `FULL_SCREEN_INTERACTIVE` permet le support complet du clavier contrairement au mode `FULL_SCREEN` qui se limite à quelques touches utiles (comme les touches fléchées). Seulement, le support complet du clavier peut introduire des failles de sécurités (un site web imitant votre bureau en plein écran pour vous demander et voler vos identifiants par exemple), c'est pourquoi un message informatif et une demande de confirmation sont affichés à l'utilisateur si votre application est sur une page web dans un navigateur. Ce n'est pas le cas pour le mode `FULL_SCREEN` qui restreint l'utilisation du clavier.

Avant de nous lancer tête baissé, réfléchissons un peu et analysons ce que nous voulons faire. Grâce aux propriétés `keyCode` et `shiftKey`, il est possible de détecter la combinaison des touches Shift et Entrée. En revanche dans ce cas là, la gestion de l'évènement qui engendrera le basculement en plein écran sera faite lors de l'enfoncement de la touche Entrée. C'est pourquoi à ce moment, nous n'aurons pas accès à la propriété `keyLocation` généré l'enfoncement de la touche Shift. Ainsi pour pouvoir gérer correctement ce cas de figure, nous serons dans l'obligation de déclarer une variable chargée de conserver la valeur de la propriété `keyLocation` de l'évènement précédent.

Pour commencer, nous pouvons donc mettre en place l'écouteur suivant :

Code : Actionscript

```
var lastKeyLocation:uint = 0;
stage.addEventListener(KeyboardEvent.KEY_DOWN, changerDAffichage);
function changerDAffichage(event:KeyboardEvent):void
{
    // Gestion du changement du mode d'affichage
    lastKeyLocation = event.keyLocation;
}
```

Ensuite, nous devons définir la condition de mise en plein écran. Pour cela, nous allons donc détecter l'enfoncement de la touche Entrée par sa propriété `keyCode` qui vaudra `Keyboard.ENTER` (équivalent à `13`), en présence de la touche Shift, et utiliser également notre variable `lastKeyLocation`, comme ceci :

Code : Actionscript

```
if (event.keyCode == Keyboard.ENTER && event.shiftKey == true &&
```

```
if (event.keyCode == Keyboard.ENTER && event.shiftKey == true &&
lastKeyLocation == KeyLocation.LEFT)
{
    // Mise en plein écran
}
```

La remise en mode d'affichage normal sera beaucoup plus simple, puisqu'il suffit de détecter une valeur `Keyboard.ESCAPE` (équivalent à `27`) de la propriété `keyCode`, qui correspond à la touche Echap.

Au final, cela nous donne le code suivant :

Code : Actionscript

```
var lastKeyLocation:uint = 0;
stage.addEventListener(KeyboardEvent.KEY_DOWN, changerDAffichage);
function changerDAffichage(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER && event.shiftKey == true &&
lastKeyLocation == KeyLocation.LEFT)
    {
        if (stage.allowsFullScreen)
        {
            // Plein écran supporté
            if (stage.allowsFullScreenInteractive)
            {
                // Plein écran avec support complet du clavier
                stage.displayState =
StageDisplayState.FULL_SCREEN_INTERACTIVE;
            }
            else
            {
                // Plein écran restreint
                stage.displayState = StageDisplayState.FULL_SCREEN;
            }
        }
    }
    if (event.keyCode == Keyboard.ESCAPE)
    {
        stage.displayState = StageDisplayState.NORMAL;
    }
    trace("Etat de l'écran : " + stage.displayState);

    lastKeyLocation = event.keyLocation;
}
```

Ne pas oublier de vérifier que les différents modes d'affichage sont disponibles à l'aide des attributs `allowsFullScreen` et `allowsFullScreenInteractive` de la classe `Stage`.

Si tout se passe bien, nous obtenons ceci dans la console en basculant entre l'affichage normal et le plein écran :

Code : Console

```
Etat de l'écran : normal
Etat de l'écran : fullScreenInteractive
Etat de l'écran : normal
```

Champs de saisie
Retour sur l'objet `TextField`

Introduction

Un champ de saisie est particulier : on y peut entrer du texte, le modifier ou l'effacer. Par exemple, vous pouvez demander le nom

de l'utilisateur grâce à un champ de saisie.
Il est possible de rendre un champ de texte éditable, pour le transformer en champ de saisie. Pour cela, rien de plus simple ! Il suffit de mettre l'attribut `type` d'un objet de la classe `TextField` à `TextFieldType.INPUT` comme ceci :

Code : Actionscript

```
monSuperTexte.type = TextFieldType.INPUT;
```

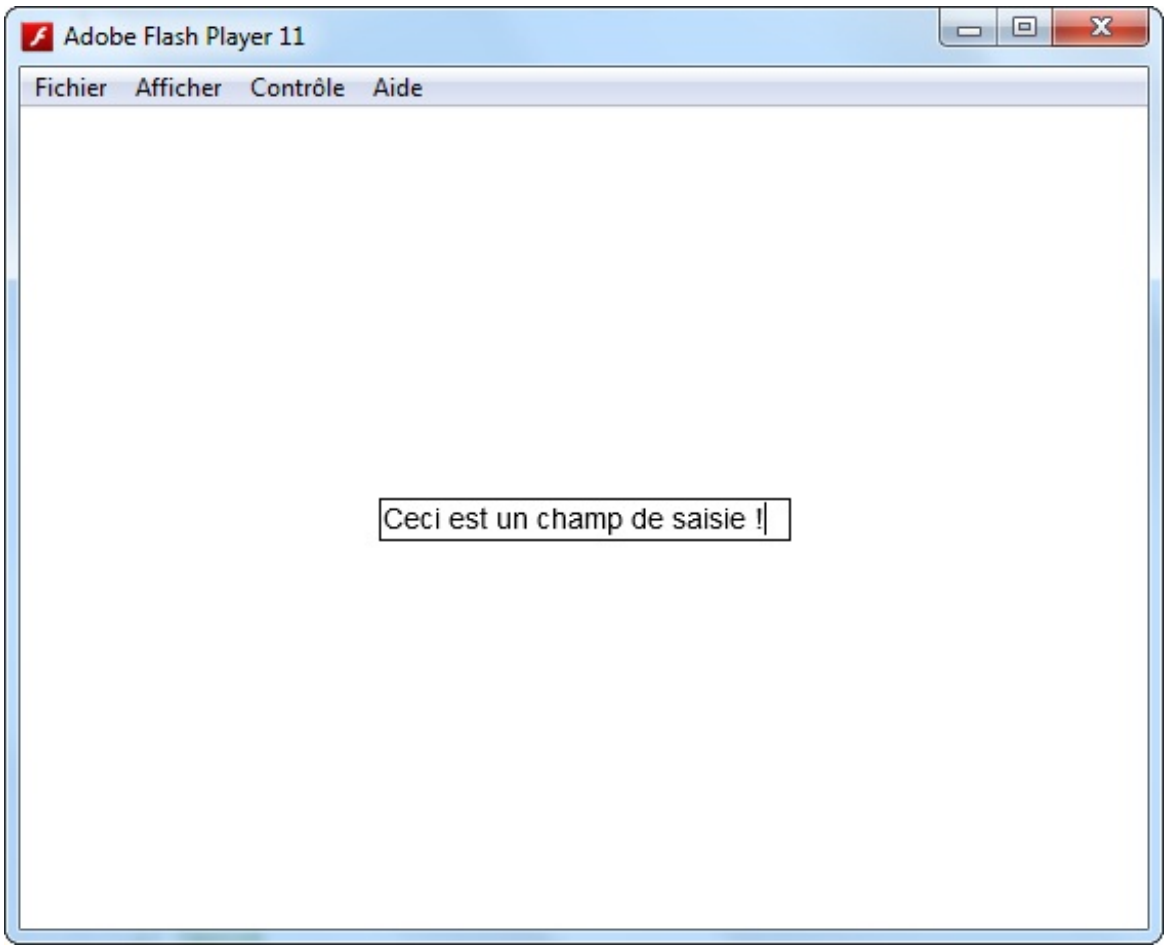
Comme d'habitude, ne pas oublier d'importer chaque classe que vous utilisez, ici `TextFieldType`.

Reprenons un code à base de `TextField` puis faisons en sorte de créer un champ de saisie :

Code : Actionscript

```
// Création de l'objet TextField
var monSuperTexte:TextField = new TextField();

// Formatage du texte
var format:TextFormat = new TextFormat("Arial", 14, 0x000000);
monSuperTexte.defaultTextFormat = format;

// Ajout d'une bordure
monSuperTexte.border = true;

// Taille
monSuperTexte.width = 200;
monSuperTexte.height = 20;

// Centrage
monSuperTexte.x = (stage.stageWidth - monSuperTexte.width) / 2;
monSuperTexte.y = (stage.stageHeight - monSuperTexte.height) / 2;

// Ajout à l'affichage
addChild(monSuperTexte);

// Il faut pouvoir sélectionner le texte (ligne falcutative)
monSuperTexte.selectable = true;

// Type du champ de texte : saisie
monSuperTexte.type = TextFieldType.INPUT;
```

Je peux maintenant écrire ce que je veux dans le champ de texte éditable :



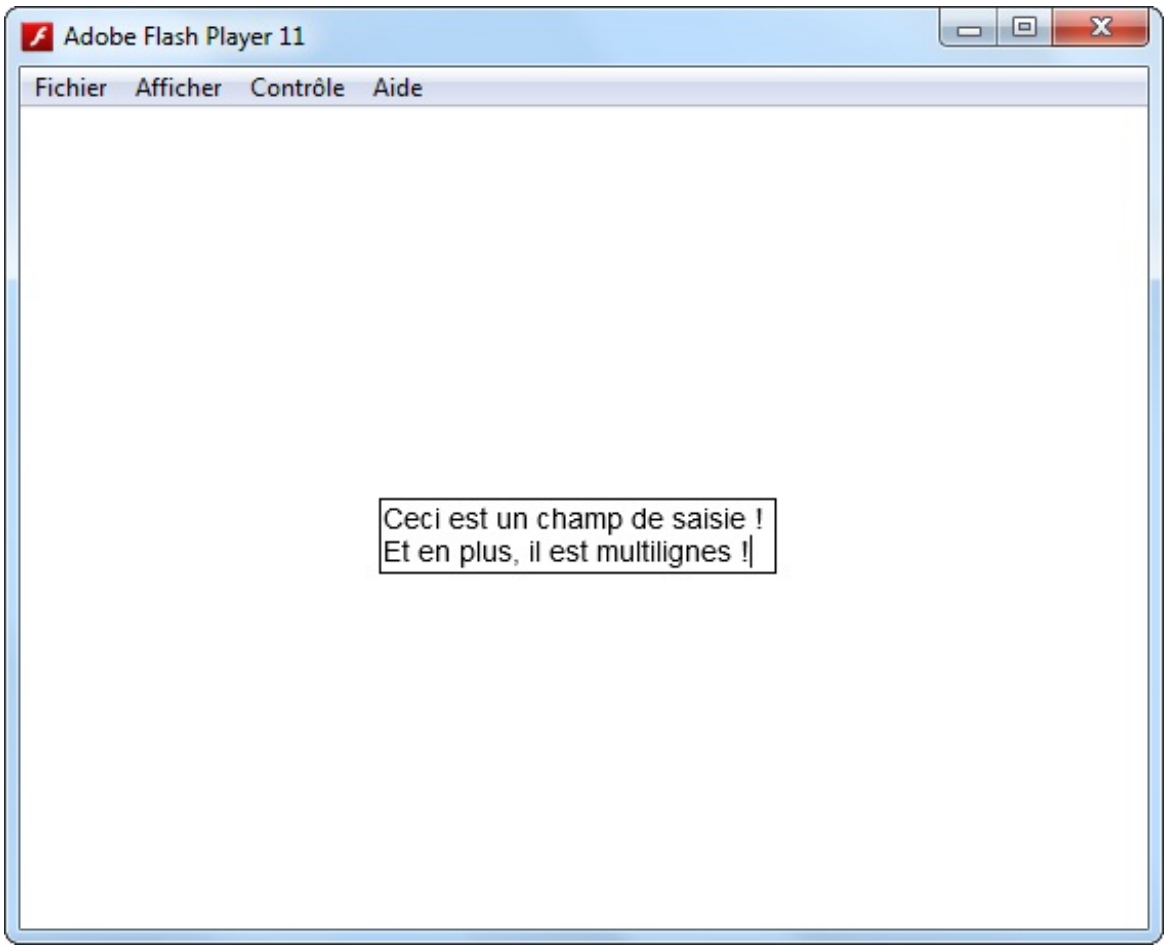
Un champ de saisie !

Bien entendu, il est tout à fait possible de rendre un champ de saisie multi-lignes. Cela fonctionne exactement de la même façon que pour un champ de texte basique :

Code : Actionscript

```
// Basculement en mode multi-lignes
monSuperTexte.multiline = true;
monSuperTexte.autoSize = TextFieldAutoSize.LEFT;
```

Ce qui nous donne la figure suivante.



Un champ de saisie multi-lignes

Restriction de la saisie

Lorsque nous définissons un objet `TextField` en tant que champ de saisie de texte, l'utilisateur peut écrire ce qu'il souhaite à l'intérieur. C'est bien, mais cela peut quand même être inapproprié dans certains cas. Imaginons que nous voulions créer un formulaire, où l'utilisateur peut renseigner des informations personnelles, telles que son nom, son prénom, sa date de naissance, etc. Il peut alors devenir gênant de se retrouver avec un nom de famille comportant des chiffres, ou encore avec une date de naissance contenant des caractères tels que « # », « @ » ou « & ».

En Actionscript, il est possible de restreindre la saisie du texte, c'est-à-dire n'autoriser la saisie que certains caractères. Pour faire ça, nous devons nous servir de la propriété `restrict` de la classe `TextField`. Il suffit alors de renseigner la liste des caractères autorisés, comme ceci :

Code : Actionscript

```
// Restriction de la saisie du texte
monSuperTexte.restrict = "ABC";
```

Une fois le programme lancé, essayez de saisir divers caractères. Vous verrez alors qu'il est possible de saisir uniquement les trois caractères que nous avons préciser au-dessus, comme le montre la figure suivante.

Restriction de la saisie aux trois caractères « A », « B » ou « C »

Je pense que certains s'imaginent déjà, qu'autoriser la saisie d'un grand nombre de caractères va être une galère monstre ! Heureusement, il existe des raccourcis pour faciliter la restriction à une plage de caractères. Par exemple, voici comment autoriser seulement la saisie de chiffres :

Code : Actionscript

```
monSuperTexte.restrict = "0-9";
```

Ce qui nous donne la figure suivante.

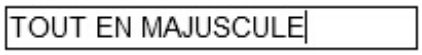
Restriction de la saisie aux caractères numériques

Pour les lettres, cela fonctionne exactement de la même manière. Voici comment faire, avec en prime l'ajout de l'espace (drôlement pratique pour séparer les mots) :

Code : Actionscript

```
monSuperTexte.restrict = "A-Z ";
```

Si vous entrez des caractères alphabétiques, ceci sont alors automatiquement saisie en majuscule comme vous voir sur la figure suivante.



Restriction de la saisie aux majuscules

Pour restreindre la saisie à l'ensemble des caractères alpha-numériques, vous devrez alors renseigner la chaîne de caractères suivante : `"A-Za-z0-9 "`.

Quelques autres propriétés utiles

Il arrive parfois que nous ayons besoin de masquer le contenu de la saisie d'un texte, à l'instar des mots de passe. Comme d'habitude, l'Actionscript a tout prévu : un attribut de la classe `TextField`, nommé `displayAsPassword`, permet de faire ceci.

Cette propriété est un booléen qu'il suffit donc de mettre à la valeur **`true`** :

Code : Actionscript

```
// Définition de la saisie en mode « mot de passe »
monSuperTexte.displayAsPassword = true;
```

À présent, si vous saisissez du texte, le contenu du champ de texte sera remplacé par une série de « * », comme sur la figure suivante.

La saisie de mot de passe

Contrairement à un champ de texte, disons traditionnel, seul l'affichage est différent. Vous pouvez ainsi utiliser la propriété `text` comme à son habitude. Cette dernière contient toujours le contenu de la saisie et non la série de « * ».

Enfin, vous pouvez également noter la présence de l'attribut `maxChars` permet de définir le nombre de caractères maximal à saisir. Une fois cette limite atteinte, plus aucun caractère ne peut être entré.

Pour l'utiliser, préciser simplement le nombre de caractères autorisé pour la saisie :

Code : Actionscript

```
monSuperTexte.maxChars = 10;
```

Évènements et `TextField`

Détecter une modification

Il est possible de détecter la moindre modification du texte contenu dans le champ de saisie, car ce dernier déclenche alors un événement de type `Event.CHANGE`. Cela se produit lorsque :

- un caractère est entré,
- un caractère est supprimé,
- du texte est collé dans le champ de saisie,
- une partie du texte est supprimée,
- une partie du texte est coupée.

De manière générale, le type d'événement `Event.CHANGE` est utilisé sur les objets qui disposent d'une valeur importante susceptible d'être modifiée : champs de saisie, barre de défilement, compteur... Vous pouvez vous-même déclencher des événements de ce type dans vos classes si besoin.

Ajoutons donc un écouteur d'événement à notre champ de saisie :

Code : Actionscript

```
monSuperTexte.addEventListener(Event.CHANGE, detecterModification);
function detecterModification(event:Event):void
{
    trace(monSuperTexte.text);
}
```

Si vous entrez du texte dans notre champ de saisie, vous obtiendrez une sortie similaire dans la console :

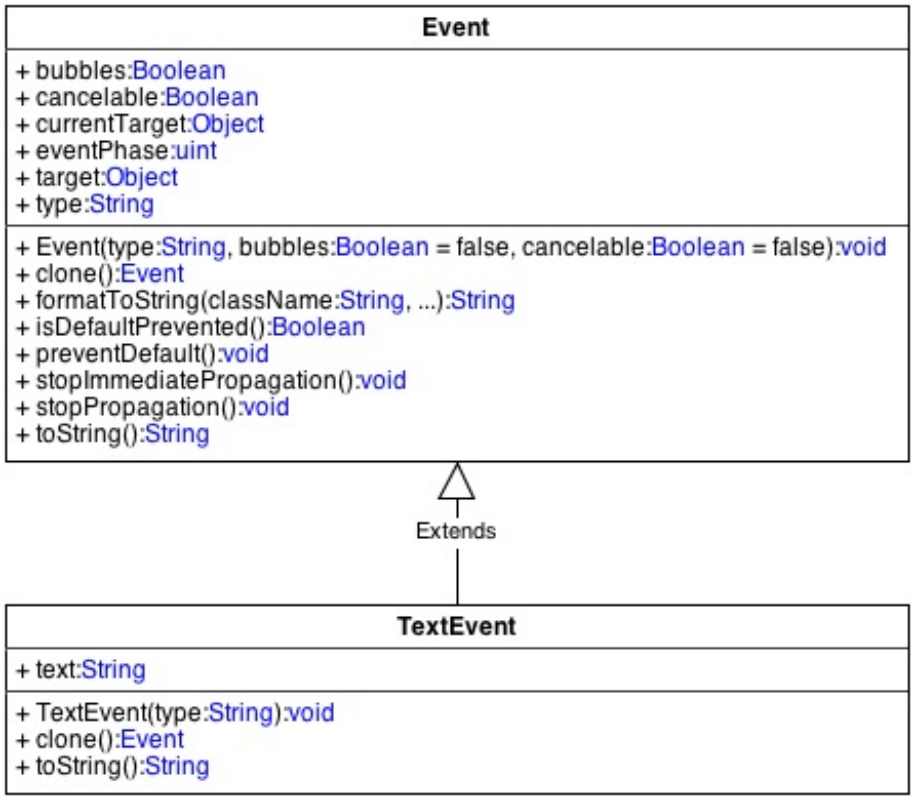
Code : Console

```
C
Ce
Cec
Ceci
Ceci
Ceci e
Ceci es
Ceci est
Ceci est
Ceci est m
Ceci est ma
Ceci est ma
Ceci est ma s
Ceci est ma sa
Ceci est ma sai
Ceci est ma sais
Ceci est ma saisi
Ceci est ma saisie
Ceci est ma saisie
Ceci est ma saisie !
```

L'objet `TextEvent`

En Actionscript, il existe un objet évènement particulièrement bien adapté aux champs de texte : il s'agit de la classe `TextEvent` !
Cet évènement sera donc étroitement lié à ce qui ce passe au niveau du clavier, mais pas uniquement. Toutefois, lors de la saisie d'un texte dans un champ à l'aide du clavier, cet évènement sera distribué, disons, en « doublon » de l'objet `KeyboardEvent`. Vous aurez alors la possibilité d'utiliser l'un ou l'autre de ces évènements, mais aussi pourquoi pas les deux !

La figure suivante reprend l'ensemble des propriétés de cette classe et de sa super-classe `Event`.



La classe TextEvent

Cette classe d'évènements ajoute donc un unique attribut nommé `text` par rapport à sa super-classe. Il vous sera donc sûrement utile !

Pour comprendre l'utilité de cet évènement, je vous propose maintenant de découvrir les deux manières dont peut être généré celui-ci. Les constantes qui servent à définir le type de ce dernier sont les suivantes :

- `LINK` : est distribué lorsque que vous cliqué sur un lien présent dans un objet `TextField`.
- `TEXT_INPUT` : est généré avant chaque modification d'un champ de saisie, donc à chaque enfoncement d'une touche, mais également lors de la répétition d'un caractère si celle-ci reste enfoncée.

L'évènement LINK est relativement simple à mettre en place. Comme nous l'avons dit, ce type d'évènements est généré lorsque vous cliqué sur un lien à l'intérieur d'un champ de texte. Souvenez-vous, pour créer un lien dans un champ de texte, nous utilisons du texte sous forme HTML. Un lien s'écrit alors de la façon suivante :

Code : HTML

```
<a href="adresse">lien</a>
```

Pour utiliser l'évènement `FOCUS`, nous devons néanmoins spécifier dans le code HTML que nous souhaitons faire appel à cet évènement. Pour cela, il est nécessaire de rajouter la chaîne de caractères « event: » en entête du lien, comme cela :

Code : Actionscript

```
monSuperTexte.htmlText = "<a href='event:adresse'>lien</a>";
```

Ensuite, nous pouvons définir notre fonction d'écouteur, comme vous savez maintenant le faire :

Code : Actionscript

```
monSuperTexte.addEventListener(TextEvent.LINK, capterLien);
function capterLien(event:TextEvent):void
{
    trace(event.text);
}
```

En cliquant sur le lien dans Flash Player, vous pourrez alors voir apparaître ceci dans la console de sortie :

Code : Console

```
adresse
```

Le second évènement proposé par la classe `TextEvent` est donc `TEXT_INPUT` !
Pour commencer, je vous invite à tester l'écouteur suivant, afin de bien voir comment Flash Player réagit à cet évènement :

Code : Actionscript

```
monSuperTexte.addEventListener(TextEvent.TEXT_INPUT,
detecterSaisie);
function detecterSaisie(event:TextEvent):void
{
    trace(event.text);
}
```

Essayez alors de saisir quelques caractères au clavier :

Code : Console

```
C
e
c
i

e
s
t

m
a

s
a
i
s
i
e

!
```

Quel est l'intérêt de cet évènement ? Ne pouvons-nous pas faire la même chose avec `KeyboardEvent.Key_DOWN` ?

Il est vrai que vu d'ici, ce type d'évènements n'apporte rien de plus par rapport à un évènement issus de la classe `KeyboardEvent`. Néanmoins pour juger de son utilité, il est nécessaire de bien comprendre la différence de génération de ces deux évènements. Le premier est généré dès lors qu'une touche est enfoncée, quelle qu'elle soit. En revanche, le second est distribué uniquement dans le cas d'une entrée de texte. Dans ce dernier cas, nous pouvons donc exclure l'appui sur les touches suivantes : Alt, Ctrl, Shift, Tab, ou encore Entrée. Par ailleurs, la classe `TextEvent` facilite énormément la gestion du texte. Par exemple, ici, pas question d'encodage ! La propriété `text` contient directement le caractère et non son code numérique.

Enfin, une dernière petite subtilité rend l'utilisation de cet évènement très pratique. Il faut savoir que ce n'est qu'une fois que l'évènement `TextEvent.TEXT_INPUT` a fini son aller-retour dans le flux d'évènements, que le champ de saisie est mis à jour. Ceci va donc nous permettre de revenir sur des propriétés de la super-classe `Event`. Rappelez-vous, cette classe disposait d'un attribut nommé `cancelable`. Et bien, écoutez bien ! Cet attribut indique si le comportement associé à l'événement peut être évité. Dans notre cas, il s'agit de la mise à jour du contenu du champ de texte actif. Si ce comportement par défaut peut être évité,

la méthode `preventDefault()` permet quant à elle de l'annuler.

Essayer donc le code suivant :

Code : Actionscript

```
monSuperTexte.addEventListener(TextEvent.TEXT_INPUT,
detecterSaisie);
function detecterSaisie(event:TextEvent):void
{
    if (event.cancelable == true) {
        trace(event.text);
        event.preventDefault();
    }
}
```

Voyez plutôt le résultat, plus aucun caractère n'est affiché à l'intérieur du champ de saisie de texte.

La gestion du focus

Dans une interface graphique complexe, composée par exemple de plusieurs champs de saisie, un seul objet peut être actif à la fois. J'entends par là qu'un seul champ de texte doit être rempli lorsque vous tapez au clavier. Cela introduit donc la notion de **focus**, qui désigne alors l'objet cible actif. Pour repérer cet objet d'affichage au sein de l'ensemble de la liste d'affichage, une référence à ce dernier est stocké dans l'attribut `focus` de `stage`.
Nous avons accès à cet attribut par un getter et un setter. Il est donc tout à fait possible de définir nous-mêmes l'objet qui a le focus.

Lorsque le focus est modifié, un évènement de type `FocusEvent` est alors distribué. Pour un objet donné, deux types d'évènements sont distincts :

- `FOCUS_IN` : généré lorsque que l'objet interactif prend le focus.
- `FOCUS_OUT` : généré lorsque que l'objet interactif perd le focus.

Sachant que l'exercice qui suit utilise ces types d'évènements, nous nous passerons d'exemples ici.

Exercice : un mini formulaire

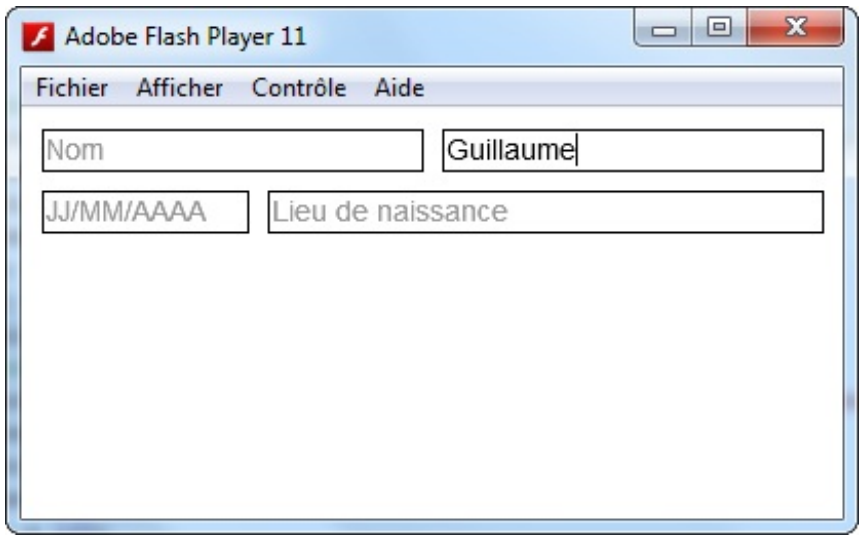
Pour conclure ce chapitre, je vais vous présenter un petit exercice consistant à réaliser un formulaire. Ce sera l'occasion de revenir sur certains points important, tout en utilisant les évènements fournis par Flash Player.

Comme je l'ai dit, l'objectif est de réaliser un mini-formulaire. Nous utiliserons quatre champs de saisie de texte afin de pouvoir renseigner les informations suivantes, pour un quelconque traitement :

- nom,
- prénom,
- date de naissance,
- lieu de naissance.

Une particularité de notre formulaire sera de présenter les intitulés, ou étiquettes, à l'intérieur des champ de saisie eux-mêmes. Pour être plus précis, le concept est de pré-remplir les champs de texte par leur étiquette. Puis nous utiliserons les évènements `FOCUS_IN` et `FOCUS_OUT` pour effacer ou remettre en place ces intitulés.

Pour mieux visualiser la chose, voici un aperçu de ce que nous obtiendrons à la fin de l'exercice :



Un formulaire avec libellé intégré

Étant donné que nous serons amenés à utiliser plusieurs champs de saisie similaires, il peut être intéressant de définir une nouvelle classe. Je suggère donc de créer un objet d'affichage personnalisé à partir d'un `TextField`.

Dans un premier temps, je vous propose de découvrir cette classe que j'ai nommée `EditableTextField`, puis nous l'analyserons ensuite :

Code : Actionscript - EditableTextField

```
package
{
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.text.TextFieldType;
    import flash.events.FocusEvent;

    public class EditableTextField extends TextField
    {
        // Attributs
        private var _etiquette:String;
        private var _formatEtiquette:TextFormat;
        private var _formatEdition:TextFormat;

        public function EditableTextField(etiquette:String)
        {
            _etiquette = etiquette;
            _formatEtiquette = new TextFormat("Arial", 14,
0x999999);
            _formatEdition = new TextFormat("Arial", 14, 0x000000);
            defaultTextFormat = _formatEtiquette;
            type = TextFieldType.INPUT;
            selectable = true;
            border = true;
            text = _etiquette;
            addEventListener(FocusEvent.FOCUS_IN, effacerEtiquette);
            addEventListener(FocusEvent.FOCUS_OUT,
remettreEtiquette);
        }

        // Méthodes d'écouteurs
        private function effacerEtiquette(event:FocusEvent):void
        {
            if (text == _etiquette)
            {
                defaultTextFormat = _formatEdition;
                text = "";
            }
        }

        private function remettreEtiquette(event:FocusEvent):void
```

```
            {
                if (text == "")
                {
                    defaultTextFormat = _formatEtiquette;
                    text = _etiquette;
                }

            }

        }

    }
```

Pour analyser cette classe, prenons les choses telles qu'elle viennent. En premier, j'ai déclaré les trois attributs suivants :

- `_etiquette` : contient l'intitulé ou l'étiquette du champ de texte,
- `_formatEtiquette` : définit le formatage à utiliser lors de l'affichage de l'intitulé,
- `_formatEdition` : définit le formatage à adopter lors de l'édition du champ de texte.

À l'intérieur du constructeur de la classe, il n'y a rien de bien compliqué. Nous initialisons l'ensemble des attributs, et en profitons pour regrouper les différentes instructions communes, touchant aux propriétés de la super-classe `TextField`. Pour clôturer ce constructeur, deux écouteurs sont mis en place afin de gérer la transition entre l'affichage de l'étiquette et son effacement avant l'édition du champ de saisie. Je ne vais entrer plus dans les détails, car je juge que vous êtes à présent suffisamment doués pour comprendre le fonctionnement de cette classe tous seuls.

En définissant les écouteurs de cette manière, à l'intérieur d'une classe, vous pouvez facilement simplifier et alléger le code. Effectivement, si vous aviez dû le faire depuis l'extérieur de la classe, vous auriez très certainement défini deux fonctions d'écouteurs pour chacun des champs de saisie de l'interface graphique. N'hésitez pas à utiliser cette technique pour simplifier et organiser au mieux votre code.

Maintenant, il ne reste plus qu'à disposer les champs de texte à l'écran, et à ajuster les derniers détails :

Code : Actionscript

```
// Champ de saisie du nom de famille
var nom:EditableTextField = new EditableTextField("Nom");
nom.restrict = "A-Za-z\\- ";
nom.width = 185;
nom.height = 20;
nom.x = 10;
nom.y = 10;
addChild(nom);

// Champ de saisie du prénom
var prenom:EditableTextField = new EditableTextField("Prénom");
prenom.restrict = "A-Za-z\\- ";
prenom.width = 185;
prenom.height = 20;
prenom.x = 205;
prenom.y = 10;
addChild(prenom);

// Champ de saisie de la date de naissance
var dateDeNaissance:EditableTextField = new
EditableTextField("JJ/MM/AAAA");
dateDeNaissance.restrict = "0-9/";
dateDeNaissance.maxChars = 10;
dateDeNaissance.width = 100;
dateDeNaissance.height = 20;
dateDeNaissance.x = 10;
dateDeNaissance.y = 40;
addChild(dateDeNaissance);

// Champ de saisie du lieu de naissance
var lieuDeNaissance:EditableTextField = new EditableTextField("Lieu
```

```
        de naissance");
        lieuDeNaissance.restrict = "A-Za-z0-9\\- ";
        lieuDeNaissance.width = 270;
        lieuDeNaissance.height = 20;
        lieuDeNaissance.x = 120;
        lieuDeNaissance.y = 40;
        addChild(lieuDeNaissance);
```

Vous remarquerez à travers cet exemple, que la définition de la propriété `restrict` n'affecte que la saisie de texte depuis l'extérieur du programme. Cela explique pourquoi l'affichage de l'intitulé de l'objet `dateDeNaissance` ne pose aucun problème.

En résumé

- Les actions produites par l'utilisateur sur la souris génèrent des objets de la classe `MouseEvent`.
- L'affichage de la souris peut être géré via les méthodes de la classe `Mouse`.
- Le glisser-déposer est une technique qui peut être contrôlé par les méthodes `startDrag()` et `stopdrag()`.
- Un objet `KeyboardEvent` est généré après toute action produite sur le clavier.
- Un objet `TextField` est transformable en champ de saisie, en redéfinissant la valeur de la propriété `type` à l'aide les constantes de la classe `TextFieldType`.
- Différentes propriétés de la classe `TextField` servent à paramétrer la saisie d'un texte, telles que `restrict`, `displayAsPassword` ou `maxChars`.
- Les évènements issus des classes `TextEvent` et `FocusEvent` augmentent les possibilités de la gestion d'évènements provenant de la souris ou du clavier.

Les collisions

Un jour, si êtes amenés à programmer des jeux vidéo, vous allez devoir vous intéresser à la notion de collisions. Vous souhaiterez alors savoir si votre voiture quitte la route, si un objet sort du cadre de la scène, si deux billes se touchent, ou encore si votre personnage a les pieds sur terre...

Au cours de ce chapitre, nous découvrirons ensemble les prémices de la théorie des collisions. Ainsi, nous serons capables de détecter une collision entre des objets de différentes tailles et différentes formes. Nous verrons également comment affiner, plus ou moins, cette détection de collisions, en fonction de la forme des objets, afin d'optimiser au maximum les performances du programme.

À la fin de ce chapitre, vous serez enfin en mesure de réaliser vos propres jeux vidéo !

Préambule

Définition d'une collision

Comme cela a été dit en introduction, nous parlerons de **collisions** tout au long de ce chapitre.
Aussi, pour mieux comprendre tout ce qui va suivre, nous allons redéfinir proprement le terme « collision ».

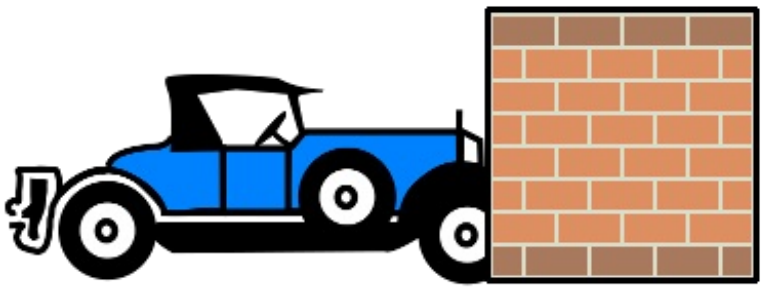
Dans la vie de tous les jours, nous attribuons ce terme à toutes sortes d'impact ou de choc entre deux objets. Nous pouvons alors citer l'exemple d'un verre qui se brise au contact du sol, d'une balle qui atteint sa cible, ou encore d'une voiture qui percute un mur.

Une voiture ayant visiblement subit une collision

Dans un programme, nous pourrons parler de collision lorsque deux objets « se touchent » ou se chevauchent. De la même manière, nous pourrons savoir si un objet `verre` est en contact avec un autre objet `sol`, ou si notre objet `voiture` percute l'objet `mur`.

De base, rien n'empêche votre objet `voiture` de continuer sa course à travers l'objet `mur`. Cela ne posera absolument aucun problème à votre application pour continuer à s'exécuter. Il faut avouer que c'est tout de même fâchant !
Il nous incombe donc de gérer les collisions à l'intérieur de notre programme, afin de pouvoir exécuter des tâches appropriées.

Sur l'image suivante, vous pouvez voir que les objets `voiture` et `mur` se chevauchent légèrement, et sont par conséquent en collision.



Collision entre les objets voiture et mur

Détecter des collisions

Des fonctions booléennes

Le principal objectif pour un programmeur, sera donc de *détecter* l'existence ou non d'une collision entre deux objets.

Mais alors, comment savoir s'il y a collision entre deux objets ?

C'est justement toute la difficulté de la chose, mais également la raison d'être de ce chapitre. Nous verrons qu'il existe différents moyens d'y arriver, qui dépendent principalement de la forme des objets et de la précision souhaitée.

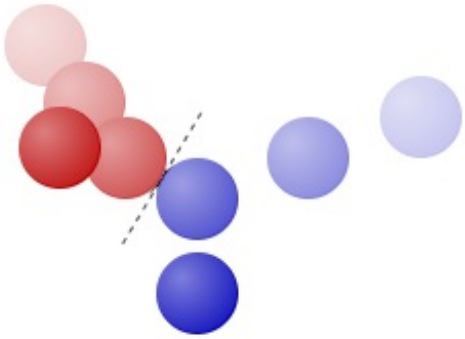
Quoi qu'il en soit, la détection de collisions se fera toujours de la même manière en Actionscript, comme dans les autres langages. Le principe est d'utiliser diverses fonctions ou méthodes, prenant en paramètres les objets à tester, et renvoyant un booléen. Ce booléen représentera alors l'existence ou non d'une collision entre les objets transmis.

L'intégralité de la gestion des collisions s'appuiera donc uniquement sur ces fonctions.

Effets produits par une collision

Les effets engendrés par une éventuelle collision, ne sont pas pris en compte dans ces fonctions de détection de collisions. Ce n'est qu'une fois la collision confirmée qu'il est possible de gérer l'impact généré par celle-ci.

Prenons l'exemple d'un choc entre deux boules quelconques, comme sur l'image suivante.



Choc élastique entre deux boules

Dans l'exemple précédent, nous voyons que nos boules sont en collision à la troisième position. À la suite de ce choc, les boules sont alors déviées et leur vitesse modifiée. Dans ce cas précis, nous pourrions nous appuyer sur la *théorie des chocs élastiques* pour mettre à jour le mouvement de chacune des boules. Néanmoins l'effet produit par la collision aurait été traitée différemment s'il s'agissait d'un choc entre un objet `voiture` et un objet `mur`, ou encore entre les objets `verre` et `sol`.

Les instructions qui découlent d'une collision sont donc directement liés aux types d'objets que vous manipulez, ou plutôt à ce qu'ils représentent. Par ailleurs, cela dépend également de ce que vous, en tant que programmeur, souhaitez faire. Par exemple, l'impact de l'objet `voiture` sur le `mur` pourrait stopper net cette dernière en déformant son capot, mais pourrait tout aussi bien la faire « rebondir » et laisser celle-ci intacte. De plus, suivant la vue utilisée (vue de face, de dessus, de côté, etc.) et la manière dont vos classes sont construites, les instructions ne seraient pas tout les mêmes.

Les manipulations post-collision sont spécifiques à chaque programme, et ne sont pas forcément transposables pour

tous vos projets. Dans ce chapitre, nous nous donc concentrerons uniquement sur les méthodes de détection de collisions entre objets.

La théorie des collisions

Collisions rectangulaires

La méthode `hitTestObject()`

Par défaut, tout objet héritant de `DisplayObject` dispose de deux méthodes servant à détecter des collisions entre objets d'affichage. La première, nommée `hitTestObject()`, est celle dont nous allons parler maintenant. Mais pour cela, nous aurons besoin de deux objets.
Traçons deux rectangles qui se superposent, au moins partiellement :

Code : Actionscript

```
var obj1:Shape = new Shape();
obj1.graphics.beginFill(0x000000);
obj1.graphics.drawRect(0, 0, 100, 100);
var obj2:Shape = new Shape();
obj2.graphics.beginFill(0x880088);
obj2.graphics.drawRect(50, 50, 100, 100);
addChild(obj1);
addChild(obj2);
```

L'image suivante nous confirme que les objets `obj1` et `obj2` se chevauchent bien.

Collision entre deux rectangles

À présent testons la collision entre ces deux objets.

Pour cela, nous allons donc nous servir de la méthode `hitTestObject()`, qui prend en paramètre, simplement l'objet d'affichage avec lequel tester la collision. Sachant que la méthode `hitTestObject()` appartient à la classe `DisplayObject`, nous pouvons aussi bien l'utiliser à partir de l'objet d'affichage `obj1` comme `obj2`.

Pour s'assurer de son bon fonctionnement, affichons le résultat de cette fonction dans la console, comme ceci :

Code : Actionscript

```
trace(obj1.hitTestObject(obj2)); // Affiche : true
```

Comme vous n'êtes pas dupes, confirmons le bon fonctionnement de la méthode `hitTestObject()`, en déplaçant un de des rectangles et en ré-affichant le résultat :

Code : Actionscript

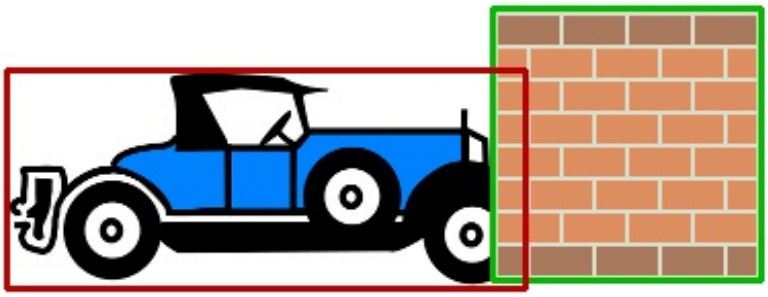
```
obj2.y = 100;
trace(obj1.hitTestObject(obj2)); // Affiche : false
```

La méthode `hitTestObject()` possède néanmoins un énorme défaut lorsqu'il s'agit d'étudier un collision entre deux objets non rectangulaires. En effet, la détection de collision réalisée à l'intérieur de cette méthode est réalisée non pas sur les objets d'affichages eux-mêmes, mais sur leur **cadre de sélection**, dont nous allons parler maintenant.

Les cadres de sélection

Un **cadre de sélection** (ou **bounding box** en anglais) peut être défini comma la plus petite zone rectangulaire incluant intégralement un objet d'affichage.

Étant donné qu'un dessin est toujours plus parlant, reprenons l'exemple de la collision entre nos objets d'affichage `voiture` et `mur`. L'image suivante nous montre alors ces deux objets délimités par leur cadre de sélection respectif.



Les objets d'affichage et leur cadre de sélection

La classe `DisplayObject` dispose des deux méthodes `getBounds()` et `getRect()` pour récupérer les coordonnées du cadre de sélection d'un objet d'affichage. La différence entre celles-ci est que la méthode `getRect()` définit le cadre de sélection sans tenir compte des lignes contenues dans l'objet d'affichage. Pour plus d'informations, je vous invite à visiter cette page.

Pour revenir aux collisions rectangulaires, la méthode `hitTestObject()` réalise donc un test de collision entre les cadres de sélection de deux objets d'affichage, et non sur leurs formes réelles.

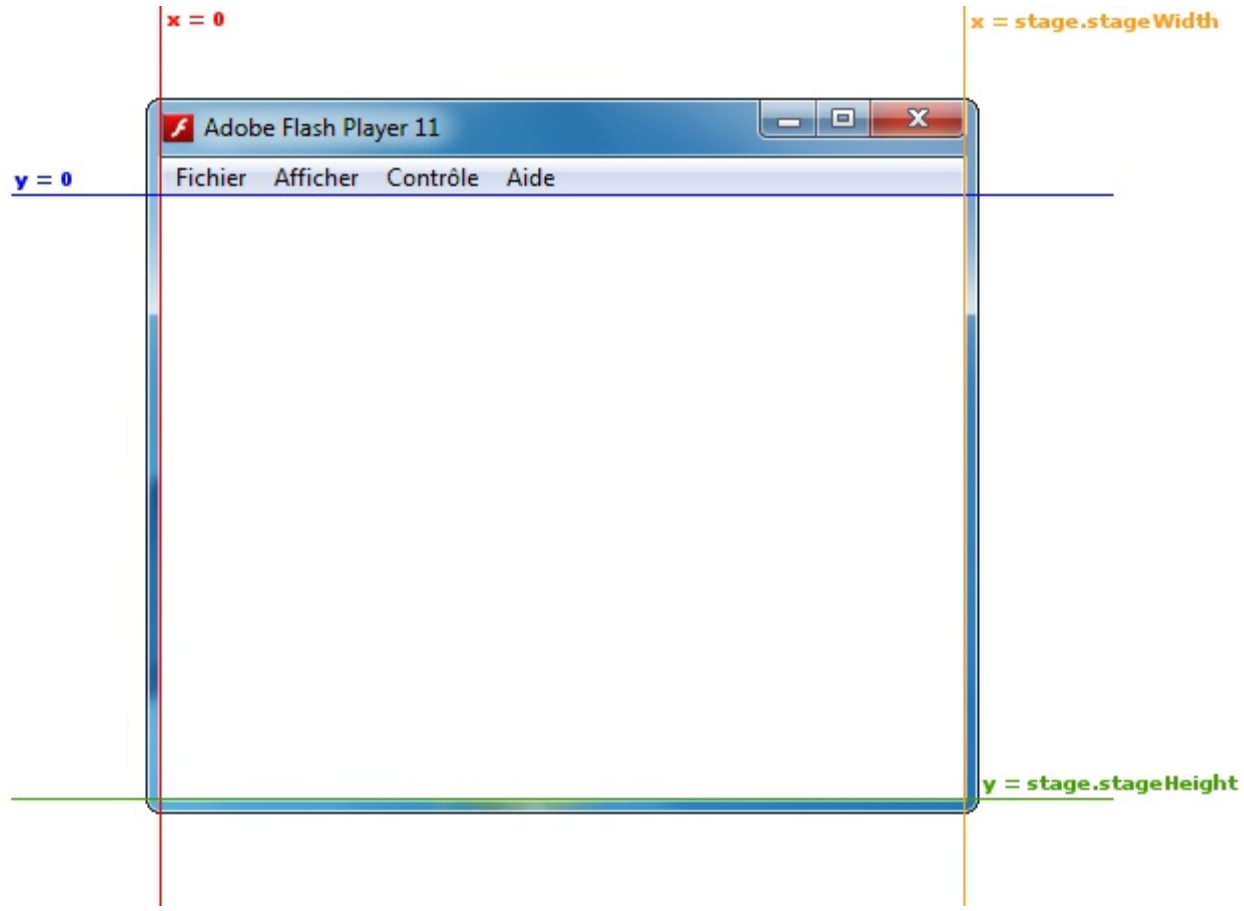
Ce type de collisions peut être donc être utile pour tout objet de forme rectangulaire, bien entendu, mais pas que ! La méthode `hitTestObject()` peut aussi vous servir lorsque vous n'avez pas besoin d'une précision extrême. Et enfin, les détections de collisions rectangulaires peuvent également être suffisante lorsque vos objets non rectangulaires sont de tailles minimes. En effet, pour des objets très petits, le cadre de sélection est quasiment confondu avec l'objet lui-même.

Collisions avec la scène principale

Jusqu'à présent, nous avons vu uniquement comment détecter des collisions rectangulaires correspondant, plus ou moins, à deux objets qui se percutent. Toutefois dans d'autres cas, il peut être utile de détecter si un objet sort d'une certaine zone délimitée.

Par exemple, ce type de collisions peut être utilisé pour détecter si un objet sort de la scène principale. Nous allons donc voir maintenant comment gérer ce genre de cas.

Sur l'image suivante, j'ai rappelé les coordonnées limites de la scène principale de votre animation.



Limites de la scène principale

Pour apprendre à gérer les collision avec la scène principale, je vous propose un petit exemple. Nous allons ainsi réaliser une animation, où un rectangle suivra les mouvements de souris, mais sans sortir du cadre de la scène principale.

Commençons par dessiner un rectangle et faisons-le suivre la souris. Vous devriez maintenant savoir coder tout ça, mais je vais quand même vous donner de quoi suivre au cas où :

Code : Actionscript

```
var rectangle:Shape = new Shape();
rectangle.graphics.beginFill(0x00BB00);
rectangle.graphics.drawRect(-50, -50, 100, 100);
addChild(rectangle);
stage.addEventListener(MouseEvent.MOUSE_MOVE, deplacer);
function deplacer(event:MouseEvent):void
{
    rectangle.x = event.stageX;
    rectangle.y = event.stageY;
}
```

Pour tester si notre objet sort de la scène principale, nous devons tester les limites de notre rectangle par rapport aux dimensions de l'animation. Étant donné que notre rectangle est centré sur son origine locale, nous pouvons détecter les dépassements de cette manière :

Code : Actionscript

```
function deplacer(event:MouseEvent):void
{
    rectangle.x = event.stageX;
    rectangle.y = event.stageY;
    if (rectangle.x - rectangle.width / 2 < 0)
    {
        trace("Trop à gauche");
    }
```

```
        if (rectangle.y - rectangle.height / 2 < 0)
        {
            trace("Trop haut");
        }
        if (rectangle.x + rectangle.width / 2 > stage.stageWidth)
        {
            trace("Trop à droite");
        }
        if (rectangle.y + rectangle.height / 2 > stage.stageHeight)
        {
            trace("Trop bas");
        }
    }
```

Certains d'entre vous pourriez vous dire qu'ici nous n'avons pas de fonction booléenne pour la détection de collisions. Toutefois, dois-je vous rappeler d'une condition renvoie une valeur booléenne ? Nous pouvons ainsi considérer chaque condition comme une détecter de collision avec une bordure. Dans cet exemple, Nous avons donc quatre tests de collisions différents ; évidemment, puisqu'un rectangle compte quatre côtés !

Enfin, pour empêcher le rectangle de sortir de la scène principale, il suffit de réinitialiser sa position pour que celui-ci reste adjacent à la bordure éventuellement franchie.

Voilà comment nous pourrions faire :

Code : Actionscript

```
    function deplacer(event:MouseEvent):void
    {
        rectangle.x = event.stageX;
        rectangle.y = event.stageY;
        if (rectangle.x - rectangle.width / 2 < 0)
        {
            rectangle.x = rectangle.width / 2;
        }
        if (rectangle.y - rectangle.height / 2 < 0)
        {
            rectangle.y = rectangle.height / 2;
        }
        if (rectangle.x + rectangle.width / 2 > stage.stageWidth)
        {
            rectangle.x = stage.stageWidth - rectangle.width / 2;
        }
        if (rectangle.y + rectangle.height / 2 > stage.stageHeight)
        {
            rectangle.y = stage.stageHeight - rectangle.height / 2;
        }
    }
```

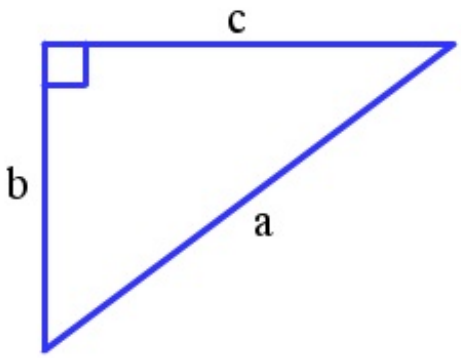
Ici, nous avons travaillé avec la scène principale, néanmoins il est tout à fait possible de travailler avec un conteneur enfant de type `Sprite`. Il faudrait alors utiliser l'espace de coordonnées local à ce conteneur, ainsi que ses dimensions décrites par les attributs `width` et `height`. Pour généraliser ceci, quelque soit la position du rectangle par rapport à sa propre origine, il est possible d'utiliser la méthode `getBounds()` de l'objet en question.

Collisions circulaires

Le théorème de Pythagore

Pour détecter des collisions entre deux objets de forme circulaire, nous aurons besoin de calculer la distance qui sépare leur centre. Pour cela, nous utiliserons le *théorème de Pythagore* !
Certains d'entre vous auront peut-être l'impression de retourner sur les bancs de l'école, mais il est important de faire un petit rappel sur ce théorème. Je passerai néanmoins relativement vite sur ce point.

Pour rappel, le théorème de Pythagore définit une relation entre les trois côtés d'un *triangle rectangle*, tel que celui illustrer sur l'image suivante.



Un triangle rectangle

En utilisant la notation de l'image précédente, le théorème de Pythagore définit la relation suivante :

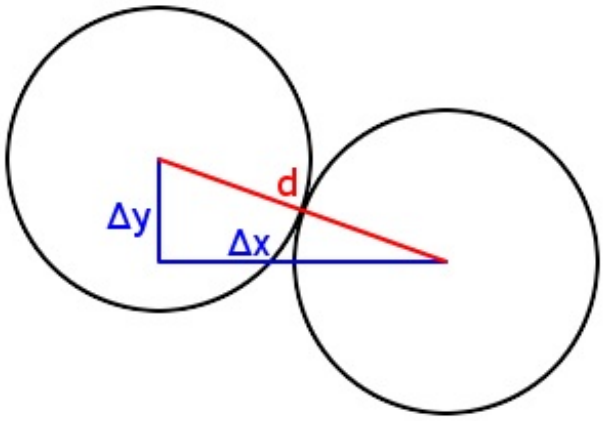
$$a^2 = b^2 + c^2$$

D'un point de vue littéraire, ce théorème est définit de la manière suivante : « si un triangle est rectangle, alors le carré de l'hypoténuse est égale à la somme des carrés des deux autres côtés ».

Calcul des distances

Maintenant, nous allons voir comment calculer la distance entre deux objets. Dans le cas présent, nous nous focaliserons sur les cercles. Néanmoins ceci peut facilement être transposables à d'autres types d'objets.

Lorsque nous utilisons le terme « distance » entre deux objets, nous parlons de celle qui sépare les centres de nos objets. Ainsi, sur l'image suivante, nous pouvons voir comment elle est défini dans le cas des cercles.



Distance entre deux cercles

D'après le théorème de Pythagore, nous avons donc la relation suivante :

$$d^2 = (\Delta x)^2 + (\Delta y)^2$$

Dans l'expression précédente, Δx représente la différence d'abscisse entre les deux objets, c'est-à-dire $x_2 - x_1$. Bien entendu, c'est la même chose pour y.

Nous supposerons ici que les objets d'affichage correspondant aux cercles, possèdent leur origine d'affichage au centre des cercles. Nous pouvons alors calculer la distance entre deux cercles de cette manière :

Code : Actionscript

```
var d:Number = Math.sqrt(Math.pow(objet2.x - objet1.x, 2) +
Math.pow(objet2.y - objet1.y, 2));
```

Toutefois, en considérant le temps nécessaire pour réaliser cette opération, il serait préférable d'utiliser quelque chose de plus

optimisé. En effet, l'utilisation de la multiplication est plus rapide que le passage par la méthode `pow()` de la classe `Math`. En fait, il serait plus intéressant, du point de vue des performances, de ne pas utiliser cette classe. C'est pourquoi, il est préférable de travailler avec des distances au carré, ce qui nous évite d'avoir recours à la méthode `sqrt()`.

Voici donc comment redéfinir cette même instruction sans utiliser la classe `Math` et en restant en distance au carré :

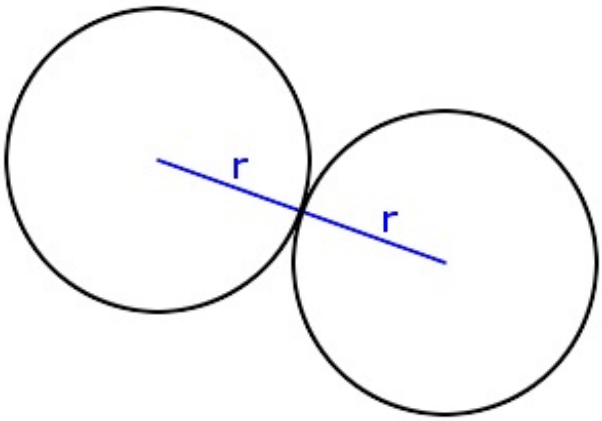
Code : Actionscript

```
var d:Number = (objet2.x - objet1.x)*(objet2.x - objet1.x) +
(objet2.y - objet1.y)*(objet2.y - objet1.y);
```

Je rappelle qu'un carré est une valeur *toujours positive*. Quel que soit les objets et le sens dont vous les utilisez, le résultat sera toujours le même.

Détecter une collision

À présent, vous savez calculer la distance entre deux cercles. Cela tombe bien, nous allons justement en avoir besoin pour détecter les collisions entre objets de forme circulaire. Quelle que soit la position et l'orientation de chacun des deux cercles, ceux-ci seront en collision si la distance entre leur centre est inférieure à une certaine valeur.
Voyez plutôt sur l'image suivante.



Distance maximale de collision

En regardant l'image précédente de plus près, nous pouvons en déduire que la distance maximale de collision entre deux cercles est $r_1 + r_2$, soit $2r$ dans le cas de deux cercles de rayons identiques. Par ailleurs, nous pouvons également utiliser le fait que le rayon d'un objet correspond exactement à la moitié de sa largeur ou de sa hauteur.

Pour détecter une collision entre deux objets d'affichage `obj1` et `obj2` de formes circulaires, nous pouvons définir une fonction de ce style :

Code : Actionscript

```
function testerCollision(obj1:DisplayObject,
obj2:DisplayObject):Boolean
{
    var collision:Boolean = false;
    var d:Number = (obj2.x - obj1.x)*(obj2.x - obj1.x) + (obj2.y -
obj1.y)*(obj2.y - obj1.y);
    if(d < (obj1.width/2 + obj2.height/2)*(obj1.width/2 +
obj2.height/2)){
        collision = true;
    }
    return collision;
}
```

Encore une fois, ce code se base sur le fait que les objets d'affichage sont centrés sur leur origine. Si ce n'est pas le cas, il sera nécessaire de l'adapter légèrement.

Collisions ponctuelles

La méthode `hitTestPoint()`

Nous allons maintenant nous pencher sur le cas des collisions ponctuelles, c'est-à-dire entre un objet d'affichage et un point. Ce dernier sera alors uniquement caractérisé par ses coordonnées `x` et `y`.

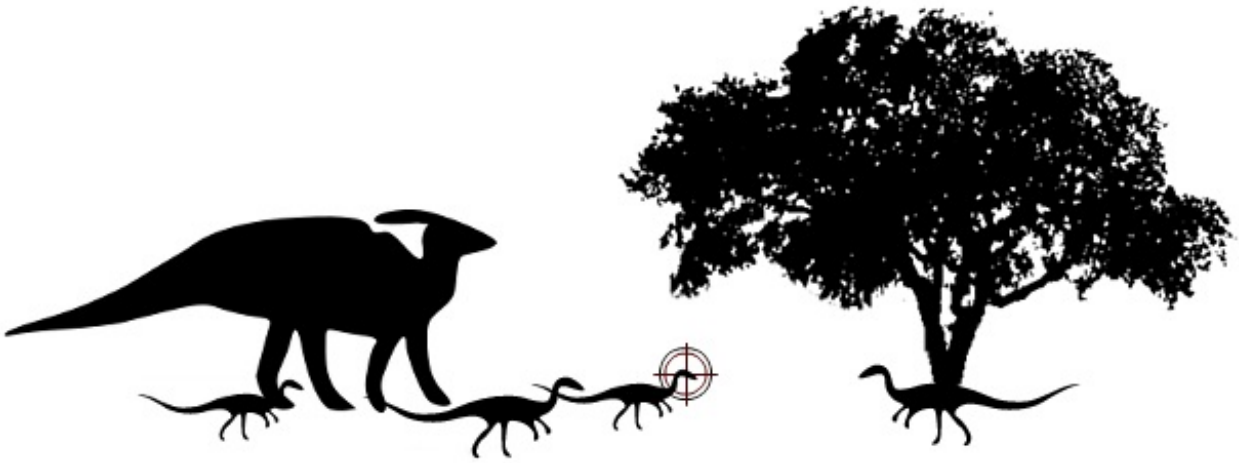
Ici, nous n'allons pas faire de théorie, mais je vais plutôt vous présenter une méthode faisant le travail à votre place. Il s'agit de la méthode `hitTestPoint()` présente dans toute classe héritant de `DisplayObject`. En voici sa signature :

Code : Actionscript

```
hitTestPoint(x:Number, y:Number, shapeFlag:Boolean = false):Boolean
```

Comme vous le voyez, l'utilisation de cette méthode nécessite de renseigner les coordonnées du point avec lequel tester la collision, ainsi d'un troisième paramètre facultatif nommée `shapeFlag`. Ce dernier paramètre, de type `Boolean`, sert à spécifier si la détection de collision doit être faite sur l'objet lui-même en tant que forme complexe (**`true`**) ou uniquement d'après son cadre de sélection (**`false`**).

Ce type de détections de collisions peut être extrêmement précieux dans certains cas. Par exemple, vous pourriez réaliser un jeu de tir et détecter la collision entre la position de votre curseur et un ennemi quelconque, comme l'illustre bien l'image suivante.



Un jeu de tir utilisant les collisions ponctuelles

Pour vous donner un exemple de code, voici comment il serait possible de réaliser cette détection à partir de la méthode `hitTestPoint()` :

Code : Actionscript

```
stage.addEventListener(MouseEvent.MOUSE_DOWN, tir);
function tir(event:MouseEvent):void
{
    if (maCible.hitTestPoint(event.stageX, event.stageY, true)) {
        trace("Cible atteinte");
    }
}
```

Si vous souhaitez tester ce code, il vous suffit d'instancier un objet d'affichage `maCible` et de l'ajouter à la liste d'affichage. Le code précédent vous servira alors à détecter les collisions entre votre curseur et cet objet, lors d'un clic avec votre souris.

Exercice : Un mini-jeu de tir

Le principe du jeu va être relativement simple. Nous allons tout d'abord dessiner un cercle à l'écran, qui baladera à l'intérieur de la

scène principale. Puis lorsque vous cliquerez dessus, votre score de points augmentera.

Démarrons tout de suite en dessinant un disque que nous placerons initialement au centre de l'écran :

Code : Actionscript

```
var cible:Shape = new Shape();
cible.graphics.beginFill(0x880088);
cible.graphics.drawCircle(0, 0, 50);
cible.x = stage.stageWidth / 2;
cible.y = stage.stageHeight / 2;
addChild(cible);
var score:int = 0;
```

Pour gérer les mouvements de notre objet d'affichage, nous allons avoir besoin d'utiliser deux variables `vitesseX` et `vitesseY` qui représenteront donc, plus ou moins, un vecteur vitesse. Pour détecter les collisions avec la scène principale, nous utiliserons le principe des collisions rectangulaires, qui dans ce cas sera plus que suffisant. À la suite d'une collision, nous pourrons mettre à jour la position de l'objet ainsi que sa vitesse, qui sera inversée suivant la composante normale à la collision.

Tout ceci peut sans doute vous paraître compliqué, pourtant, regardez le code correspondant qui n'est pas si terrible :

Code : Actionscript

```
var vitesseX:int = 1 + Math.random() * 10;
var vitesseY:int = 1 + Math.random() * 10;
addEventListener(Event.ENTER_FRAME, deplacer);
function deplacer(event:Event):void
{
    cible.x += vitesseX;
    cible.y += vitesseY;
    if (cible.x - cible.width / 2 < 0)
    {
        cible.x = cible.width / 2;
        vitesseX = - vitesseX;
    }
    if (cible.y - cible.height / 2 < 0)
    {
        cible.y = cible.height / 2;
        vitesseY = - vitesseY;
    }
    if (cible.x + cible.width / 2 > stage.stageWidth)
    {
        cible.x = stage.stageWidth - cible.width / 2;
        vitesseX = - vitesseX;
    }
    if (cible.y + cible.height / 2 > stage.stageHeight)
    {
        cible.y = stage.stageHeight - cible.height / 2;
        vitesseY = - vitesseY;
    }
}
```

Enfin, la gestion des tirs peut se faire à l'aide la méthode `hitTestpoint()`, comme cela a été décrit précédemment :

Code : Actionscript

```
stage.addEventListener(MouseEvent.CLICK, tir);
function tir(event:MouseEvent):void
{
    if (cible.hitTestPoint(event.stageX, event.stageY, true))
    {
        score += 10;
        trace("Score : " + score);
    }
```

```
        }
```

L'exercice que nous venons de réaliser est un préambule à un jeu de tir plus complexe. Cependant, la plus grosse difficulté qu'est l'interaction est plutôt bien entamée. Pour vous exercer, je vous invite donc grandement à essayer de réaliser un vrai jeu de tir suivant vos envies.

Les collisions de pixels

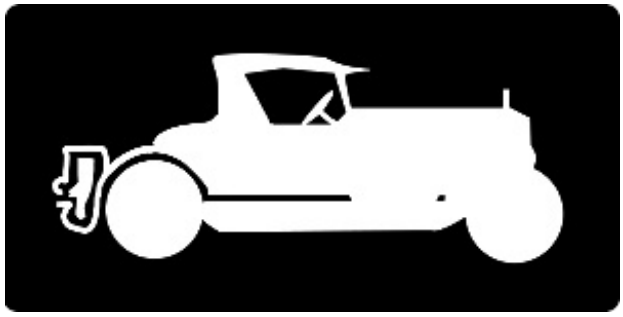
Utiliser l'opacité

Les masques

Une autre manière d'appréhender les collisions, est d'utiliser les objets d'affichage sous forme d'images bitmap. Comme vous le savez maintenant, ces images sont décrites par une série de pixels. Et rappelez-vous, les pixels sont caractérisés par des nombres de la forme `0xAARRVVBB`, où `AA` représentent l'opacité du pixel.

L'opacité d'un objet d'affichage permet justement de décrire les contours de la partie visible de celui-ci. C'est pourquoi l'opacité est grandement utile pour effectuer des tests de collisions. Mais nous reviendrons là-dessus plus tard.

En attendant, l'image suivante vous montre à quoi peut ressembler le canal `alpha` de l'image `voiture`, appelé génralement **masque** de l'image.



Masque (ou opacité) de l'objet voiture

Bien entendu, il serait tout à fait possible de réaliser des tests de collisions en utilisant les canaux rouge, vert ou bleu. Cependant, cela n'est pas très courant, aussi, nous n'en parlerons pas.

Une manière de détecter des collisions

Ici, nous allons voir comment nous pourrions procéder pour détecter une collision entre deux images bitmap. Toutefois, nous verrons que la classe `BitmapData` intègre une méthode `hitTest()`. Malheureusement je ne suis pas certain de son fonctionnement interne. Mais quoi qu'il en soit, cela ne nous empêche pas d'imaginer une manière dont cela pourrait être fait.

La technique dont je vais vous parler est simplement tirée de mon imagination. Il est donc fort probable qu'il y ait d'autres manières d'y parvenir, et de façon plus optimisée. Néanmoins, je cherche ici simplement à vous faire découvrir une méthode que vous pourriez vous-mêmes transcrire en code. Vous pourrez ensuite très facilement adapter celle-ci ou une autre à vos projets, pour des applications diverses.

La technique que je vais vous présenter maintenant, tire parti du fait qu'une transparence totale est représentée par une valeur correspondante nulle. Également, je ferais l'hypothèse que l'opacité est représentée par une valeur décimale comprise entre 0 et 1. Ainsi, en multipliant simplement les valeurs des opacités des deux pixels devant se superposer, nous obtenons un nouveau masque contenant simplement l'intersection des deux images.

Je vous propose un exemple pour illustrer ceci, utilisant des matrices. Pour cela nous utiliserons le *produit d'Hadamard* ou produit composante par composante. N'ayez pas peur, cela n'est pas si compliqué que ça en a l'air.

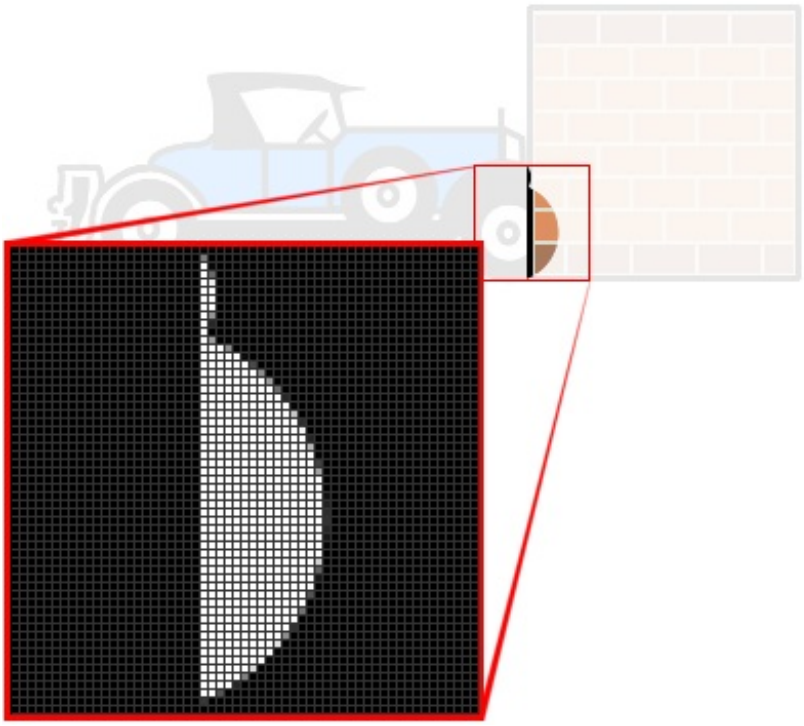
Voyez plutôt :

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Grâce à ce calcul, toute valeur non-nulle correspond donc à un pixel étant en collision entre les deux objets. Il pourrait même être

envisageable d'utiliser un seuil pour les valeurs intermédiaires entre 0 et 1, pour considérer s'il y a collision ou non.

Dans l'encadré rouge de l'image suivante, je vous laisse apprécier ce que cette méthode pourrait donner localement pour notre problème de collision entre les objets `voiture` et `mur`.



Résultat d'une multiplication entre les masques de deux images

Une fois cette multiplication faite, nous n'allons plus qu'à parcourir l'ensemble des pixels pour détecter si les pixels dépassent le seuil d'opacité définit (ou son carré si vous souhaitez prendre en compte la multiplication appliquée aux valeurs de base). Un seul pixel dépassant le seuil d'opacité est alors synonyme de collision.

Application en code

Une méthode prête à l'emploi

Vous ayant déjà détaillé le principe, je vais simplement ici vous exposer le code d'une classe intégrant une méthode statique réalisant ce travail à partir d'objets d'affichage quelconques.

Code : Actionscript

```
package
{
    import flash.display.DisplayObject;
    import flash.display.BitmapData;
    import flash.geom.Matrix;
    import flash.geom.Point;
    import flash.geom.Rectangle;

    public class Collision
    {

        public function Collision()
        {
            throw new Error('Classe abstraite');
        }

        static public function tester(obj1:DisplayObject,
obj2:DisplayObject, seuil:uint):Boolean
        {
            var collision:Boolean = false;
            if (obj1.hitTestObject(obj2) == true)
            {
```

```
                var img1:BitmapData = creerBitmapData(obj1);
                var img2:BitmapData = creerBitmapData(obj2);
                var rect1:Rectangle = obj1.getBounds(obj1);
                var rect2:Rectangle = obj2.getBounds(obj2);
                var pos1:Point = new Point(obj1.x + rect1.x, obj1.y
+ rect1.y);
                var pos2:Point = new Point(obj2.x + rect2.x, obj2.y
+ rect2.y);
                collision = img1.hitTest(pos1, seuil, img2, pos2,
seuil);
            }
            return collision;
        }

        static private function
creerBitmapData(objet:DisplayObject):BitmapData
        {
            var image:BitmapData = new BitmapData(objet.width,
objet.height, true, 0x00000000);
            var rect:Rectangle = objet.getBounds(objet);
            var transformation:Matrix = new Matrix();
            transformation.translate(-rect.x, -rect.y);
            image.draw(objet, transformation);
            return image;
        }

    }

}
```

Comme je l'ai dit avant, tout repose sur la méthode `hitTest()` de la classe `BitmapData`. Le reste est uniquement la mise en place des éléments à transmettre à cette méthode.

La méthode proposée ci-dessus possède néanmoins un petit défaut. En effet, les diverses transformations appliquées à l'objet ne sont pas prises en compte, et généreront des bugs. Les objets d'affichage utilisés doivent ne doivent donc subir aucune transformation. Néanmoins, ceci peut être corrigé en les prenant en compte lors du tracé des `BitmapData`.

Un exemple d'utilisation

Pour tous ceux qui souhaiterez voir l'efficacité de la méthode statique donnée précédemment, je vous propose ci-dessous un petit code d'exemple. Regardez alors la simplicité d'utilisation de cette méthode.
Encore une fois, un grand merci à la POO.

Code : Actionscript

```
var obj1:Shape = new Shape();
obj1.graphics.lineStyle(20, 0x000000);
obj1.graphics.drawCircle(200, 200, 100);
var obj2:Shape = new Shape();
obj2.graphics.beginFill(0x880088);
obj2.graphics.drawCircle(0, 0, 15);
obj2
addChild(obj1);
addChild(obj2);
addEventListener(Event.ENTER_FRAME, maFonction);
function maFonction():void
{
    obj2.x = mouseX;
    obj2.y = mouseY;
    if (Collision.tester(obj1, obj2, 0x11)) {
        obj1.filters = new Array(new GlowFilter());
    }else{
        obj1.filters = new Array();
    }
```

```
        }
```

Ce chapitre n'était qu'une introduction à la théorie des collisions. Même si vous savez désormais répondre à la plupart de vos besoins en termes de collisions, n'hésitez pas à vous perfectionner, notamment grâce au tutoriel « Théorie des collisions » de Fvirtman.

En résumé

- La détection d'une collision entre deux objets d'affichage se fait à l'aide d'une **fonction booléenne**.
- Suivants la forme des objets d'affichage et la précision de détection souhaitée, ces fonctions de tests de collisions doivent être différentes.
- Pour tout `DisplayObject`, la méthode `hitTestObject()` permet de détecter des collisions entre deux objets à partir de leur **cadre de sélection**.
- La méthode `hitTestPoint()` de la classe `DisplayObject` sert à tester une éventuelle collision entre un objet quelconque et un point.
- Pour affiner au mieux les collisions entre deux objets aux formes complexes, la méthode `hitTest()` de la classe `BitmapData` effectue des tests au niveau des pixels.
- Les effets engendrées par une collision doivent être traités en dehors des fonctions booléennes de détection des collisions.

La gestion des erreurs

Au cours de la prochaine partie, nous allons principalement apprendre à charger et manipuler du contenu multimédia. Nous verrons donc comment importer des données depuis l'extérieur de votre application. Ceci dit, avant d'entrer dans le vif du sujet, je vous propose ici un petit chapitre pour introduire la notion de **gestion des erreurs**.

Vous verrez que cela pourra s'avérer extrêmement utile lorsque nous travaillerons avec des données externes à l'application. Ce sera donc le cas dans la partie qui suit, puisque nous serons amenés à charger du contenu multimédia à l'intérieur de l'application. Également, cela nous servira plus tard lorsque nous traiterons des échanges de données en réseau. Aussi, essayer d'être un maximum attentif à ce qui va suivre.

Les principes de base

Introduction à la gestion des erreurs

Lorsque nous prononçons le mot « erreurs », il est fréquent de voir des sourcils se froncer. Il s'agit en effet d'un sujet délicat, qui est souvent synonyme de prises de tête. Néanmoins, nous devons passer par là pour s'assurer qu'un code s'exécute correctement.
En ce qui nous concerne, nous pouvons distinguer deux principaux types d'erreurs : les **erreurs de compilation** et les **erreurs d'exécution**.

Sans grande surprise, les **erreurs de compilation** sont des erreurs qui se produisent lors de la compilation. Elles sont généralement facile à déceler, puisque que le compilateur nous informe de leur existence. Par ailleurs, lors de l'apparition d'une erreur de ce genre, la compilation est stoppée et la génération du fichier SWF n'est pas achevée. Il nous est alors nécessaire de corriger cette erreur afin de pouvoir finaliser la compilation.
Ces erreurs ne sont cependant pas concernées par ce chapitre, mais nous nous intéresserons plutôt aux erreurs d'exécution !

Les **erreurs d'exécution**, quant à elles, surviennent comme leur nom l'indique, durant l'exécution du code. Contrairement aux erreurs de compilation, les erreurs d'exécution ne bloquent pas la compilation d'un programme, mais se produisent lors de l'exécution d'une instruction ou d'un code erroné. Suivant la complexité du programme, ces erreurs peuvent être difficiles à détecter, et donc à corriger.

La **gestion des erreurs** consiste alors à ajouter ou modifier du code dans un programme, spécialement conçu pour traiter une ou plusieurs erreurs. Cela revient à détecter les éventuelles erreurs d'exécution et à les contourner afin de pouvoir reprendre, si possible, une exécution « normale » du programme. Dans le cas échéant, il peut être intéressant d'en informer tout simplement l'utilisateur.

Les différents types d'erreurs d'exécution

Les erreurs synchrones

Les erreurs d'exécution dites « **synchrones** » se produisent lors de l'exécution d'une instruction renvoyant un *résultat immédiat*. Nous pouvons également parler d'**exceptions**. Pour vous donner un exemple, voici un code qui ajoute un objet de type `Sprite` à la liste d'affichage, sans l'avoir instancier au préalable :

Code : Actionscript

```
var objet:Sprite;
addChild(objet);
```

Comme vous pouvez le voir, la compilation du code ne pose absolument aucun problème : le fichier SWF est donc bien généré. Toutefois lors de l'exécution du code, vous allez voir apparaître ceci dans la console de sortie :

Code : Console

```
[Fault] exception, information=TypeError: Error #2007: Le paramètre child ne doit p
```

Un tel message s'affichera à chaque fois qu'une erreur n'aura pas été traitée ; on dit alors que l'exception n'a pas été interceptée. Ce message permet au développeur de cibler l'erreur afin de la *réparer* ou de la *corriger*. Comme vous avez peut-être pu le constater, l'exécution du code s'est stoppée lors de l'apparition de l'erreur. En revanche, ceci n'est valable que dans la version de débogage du Flash Player. En effet, dans la version commerciale, le Flash Player va tenter de passer outre l'erreur et de continuer

l'exécution du code si possible.

Pour en revenir à la gestion des exceptions, rappelez-vous que celles-ci sont générées dès lors que l'instruction en question est exécutée. Il est alors possible de traitée l'erreur immédiatement, dans le même bloc que l'instruction incriminée. Pour cela, nous verrons comment isolé le code erroné à l'intérieur d'une instruction **try...catch**, afin de réagir tout de suite et pouvoir reprendre l'exécution de la suite du programme.

Les erreurs asynchrones

Lors d'une opération quelconque, il peut arriver qu'une erreur ne puisse pas être retournée immédiatement. Ceci pourrait être le cas, par exemple, lors de la lecture ou de l'écriture de données. En effet, l'accès à des données externes à l'application ne se fait pas de façon immédiate. Dans cette situation, une instruction **try...catch** serait alors inutile. La réponse aux erreurs liés à ces opérations est donc gérée par événements.

Lorsque la génération d'une erreur n'est pas directe mais différée, l'erreur d'exécution est dite « **asynchrone** ». Celle-ci est représentée par la classe `ErrorEvent` ou l'une de ses sous-classes. La réponse à cette dernière se fait gra^ce à une fonction d'écouteur comme d'ordinaire.

Les erreurs synchrones

L'intruction throw

Présentation

Avant d'entamer la gestion (ou **interception**) des exceptions en elle-même, nous allons parler de l'instruction **throw**. Ce mot-clé, que nous avons déjà vaguement évoqué, sert en réalité à renvoyer une exception. Cette exception se présente alors sous la forme d'une instance de la classe `Error` ou de l'une de ses sous-classes.

Pour vous donner un exemple, voici comment renvoyer une exception :

Code : Actionscript

```
var monErreur:Error = new Error();
throw monErreur;
```

En exécutant ce code, la console de sortie laisse alors apparaître ceci :

Code : Console

```
[Fault] exception, information=Error
```

Il faut avouer que dans le cas précédent, les informations concernant l'erreur sont relativement minces. Pour remédier à ce problème, il est possible de renseigner des informations sur l'erreur dans le constructeur de la classe `Error`. Vous pouvez donc renseigner un message qui s'affichera dans la console de sortie ou qui pourra être utilisé lors de l'interception de l'erreur. Également, un second paramètre `id` peut servir à identifier avec précision l'erreur.

Cet identificateur sert généralement à rechercher les causes de l'erreur, principalement si vous n'êtes pas l'auteur de la classe l'ayant générée. Vous pouvez notamment trouver des informations précieuses suite à une recherche sur Internet.

Une erreur bien renseignée serait par exemple :

Code : Actionscript

```
var monErreur:Error = new Error("Le paramètre child ne doit pas être
nul.", 2007);
```

Exemple

À présent, nous allons réaliser une petite fonction renvoyant diverses erreurs, que nous utiliserons dans la suite avec le bloc

try...catch.

Voilà donc la fonction que je vous propose :

Code : Actionscript

```
function testerErreur(objet:Object):void
{
    if (objet == null)
        throw new TypeError("Le paramètre objet ne doit pas être
nul.", 4000);
    if (objet == stage)
        throw new Error("Le paramètre objet ne doit pas être la
scène principale.", 4001);
}
```

En l'analysant, vous constaterez que celle-ci renvoie des exceptions différentes suivant la nature de l'objet passé en paramètre. Cela pourrait correspondre par exemple à quelques vérifications nécessaire en début d'une méthode quelconque.
Vérifions tout de même le bon fonctionnement de cette fonction. En premier lieu, passons la scène principale en paramètre, et voyons comment la version de débogage du Flash Player réagit.

L'instruction correspondante est la suivante :

Code : Actionscript

```
testerErreur(stage);
```

Aucun soucis, l'exception non interceptée est bien affichée dans la console :

Code : Console

```
[Fault] exception, information=Error: Le paramètre objet ne doit pas être la scène
```

La seconde erreur renvoyée par la fonction définie plus haut, correspond au passage en paramètre d'un objet quelconque non instancié :

Code : Actionscript

```
var monObjet:Object;
testerErreur(monObjet);
```

L'erreur est également bien renvoyée et affichée dans la console de sortie :

Code : Console

```
[Fault] exception, information=TypeError: Le paramètre objet ne doit pas être nul.
```

L'instruction try...catch

Utilisation basique

Entrons maintenant dans le vif du sujet, et découvrons le fonctionnement de l'instruction **try...catch**. Cette instruction permet d'isolé le code pouvant renvoyer une exception : cela se fait alors à l'intérieur du bloc **try**. Ce bloc ne peut être utilisé

indépendamment du bloc **catch**, qui sert à intercepter les exceptions renvoyées.

Code : Actionscript

```
try {
    // Instructions à isolées
}catch(e:Error){
    // Gestion de l'exception
}
```

Par exemple, utilisons notre fonction `testerErreur()` pour comprendre comment l'ensemble **try...catch** fonctionne. En l'absence d'erreur, seul le bloc **try** est exécuté. Voyons cependant ce que cela donne lorsqu'une erreur apparaît. Tentons de renvoyer une exception de type `Error` :

Code : Actionscript

```
try {
    testerErreur(stage);
    trace("Aucune exception renvoyée");
}catch(e:Error){
    trace("Exception interceptée : #" + e.errorID);
}
```

Le résultat dans la console de sortie est le suivant :

Code : Console

```
Exception interceptée : #4001
```

Grâce à cet exemple, nous pouvons constater un certain nombre de chose. Tout d'abord, nous voyons que l'exception a bien été interceptée puisque qu'aucun message d'erreur n'est apparu dans la console de sortie. Ensuite vous remarquerez que l'exécution du bloc **try** s'est stoppée lors de l'apparition de l'erreur. Enfin, le bloc **catch** a été exécuté intégralement, et il possible de récupérer des informations

Utilisation avancée

Précédemment, nous avons vu comment mettre en place un bloc **try...catch**, permettant de gérer de façon basique les erreurs synchrones. Toutefois, il est possible de gérer de manière plus poussée les exceptions renvoyées à l'intérieur du bloc **try**.
D'une part, nous pouvons ajouter plusieurs bloc **catch**, permettant d'identifier plus spécifiquement différents types d'erreurs. Chaque instruction **catch** est alors vérifiée dans l'ordre et est associée un type d'exception. Seul le premier bloc **catch**, correspondant au type d'exception renvoyée, est exécuté.
D'autre part, le mot-clé facultatif `finally` permet d'instroduire des instructions supplémentaires, exécutées au terme de la gestion de l'erreur, quel que soit le chemin parcouru à travers les différents bloc **try** ou **catch**.

Pour illustrer tout ce qui vient d'être dit, voici un exemple de code contenant un certain nombre de blocs :

Code : Actionscript

```
try {
    testerErreur(null);
}catch (e:TypeError) {
    trace("Exception interceptée de type TypeError");
}catch(e:Error){
    trace("Exception interceptée de type Error");
}finally {
    trace("Gestion terminée");
}
```

Après exécution du code, la console laisse apparaître ceci :

Code : Console

```
Exception interceptée de type TypeError
Gestion terminée
```

Comme prévu, les blocs **try** et `finally` ont été exécutés, et le seul bloc **catch** associé au type `TypeError` a été exécuté.

Dans le cas de blocs catch associés à plusieurs type d'exceptions, ceux-ci doivent être ordonnés du type le plus spécifique au moins spécifique. Ainsi la classe `Error`, super-classe de toutes les autres, doit se être traitée en dernier. Dans l'exemple précédent, une exception de type `ArgumentError` aurait été interceptée par le dernier bloc **catch**.

Les erreurs asynchrones

Distribuer un objet `ErrorEvent`

Nous allons maintenant nous intéresser aux *erreurs asynchrones* et à leur gestion. Comme je vous l'ai introduit plus tôt, ce type d'erreurs est représenté par des événements héritant de la classe `ErrorEvent`. Pour générer une erreur asynchrone, nous pouvons donc nous servir de la méthode `dispatchEvent()` de la classe `EventDispatcher`.

Pour la suite, je vous propose donc de renvoyer des erreurs via la fonction suivante :

Code : Actionscript

```
function testerErreur(objet:Object):void
{
    if (objet == null)
        dispatchEvent(new ErrorEvent(ErrorEvent.ERROR));
}
```

Ensuite, voici comment généré ladite erreur :

Code : Actionscript

```
testerErreur(null);
```

Sans gestion directe de l'erreur, la version de débogage du Flash Player arrête l'exécution du code et renvoie l'erreur à l'intérieur de la console de sortie. Voilà le message affiché :

Code : Console

```
Error #2044: error non pris en charge : text=
```

Bien évidemment, il est préférable d'étendre la classe `ErrorEvent` afin de personnaliser l'erreur à son utilisation. D'ailleurs comme nous le verrons la prochaine partie, la classe `IOErrorEvent` est spécifique aux erreurs générés par l'interaction avec des données externes au programme.

Gérer des événements d'erreurs

Aussi surprenant que cela puisse paraître, vous n'allez rien apprendre ici. Effectivement, la gestion des événements est strictement identique, quel que soit le type d'événements. Seul la classe d'événement devra être ajuster pour pouvoir répondre aux erreurs asynchrones.

Nous pouvons donc très facilement définir une fonction de rappel pour un objet de type `ErrorEvent`, et enregistrer l'écouteur auprès de l'objet courant.
D'ailleurs en voici l'écriture :

Code : Actionscript

```
function traiterErreur(error:ErrorEvent):void
{
    trace("Erreur traitée");
}
addEventListener(ErrorEvent.ERROR, traiterErreur);
```

Testons la gestion de l'erreur comme tout à l'heure :

Code : Actionscript

```
testerErreur(null);
```

Aucune surprise, la fonction d'écouteur est exécutée lors de l'apparition de l'événement :

Code : Console

```
Erreur traitée
```

Bien comprendre les deux approches
Une classe utilisant les deux approches

Précédemment, nous avons vu comment gérer des erreurs d'exécution, qu'elles soient synchrones ou asynchrones. Cependant, il est fort probable que certains d'entre vous n'arrivent pas encore à bien distinguer les différences d'utilisation qu'il existe entre les deux types d'erreurs. Je vous propose alors ici une classe permettant d'illustrer ceci.

Présentation

Dans un premier temps, je vous laisse découvrir et vous familiariser avec le code de cette classe donné ci-dessous :

Code : Actionscript - Animation.as

```
package
{
    import flash.display.DisplayObject;
    import flash.events.ErrorEvent;
    import flash.events.Event;

    public class Animation
    {
        private var _objet:DisplayObject;
        private var _vars:Object;

        public function Animation(objet:DisplayObject = null)
        {
            _objet = objet;
            _vars = null;
        }

        public function incrementer(vars:Object):void
        {
            if (_objet == null) {
                throw new TypeError("Objet nul");
            }else {
                _vars = vars;
                _objet.addEventListener(Event.ENTER_FRAME, animer);
            }
        }
```

```
            private function animer(event:Event):void
            {
                for (var p:String in _vars) {
                try {
                    _objet[p]++;
                    if (_objet[p] >= _vars[p]) {

_objet.removeEventListener(Event.ENTER_FRAME, animer);
                    }
                }catch (e:Error) {
                    _objet.dispatchEvent(new
ErrorEvent("AnimErrorEvent"));
                }
                }
            }

        }

    }
```

Fonctionnement

Rassurez-vous, si vous n'avez pas entièrement cerner le fonctionnement de la classe `Animation`, c'est normal !
C'est pourquoi je vous propose maintenant quelques explications. Tout d'abord, et j'espère que vous l'aurez compris, cette classe va nous permettre d'animer un objet de type `DisplayObject`. Pour être plus précis, la méthode `incrementer()` va servir à lancer une animation correspondant à augmenter la valeur de certaines des propriétés de l'objet d'affichage jusqu'à une certaine valeur. Les propriétés à prendre en compte doivent être renseignées à travers le paramètre `vars`.

Revenons sur la définition de la méthode `incrementer()` :

Code : Actionscript

```
public function incrementer(vars:Object):void
{
    if (_objet == null) {
        // Résultat immédiat permettant de générer une erreur
synchrone
        throw new TypeError("Objet nul");
    }else {
        _vars = vars;
        _objet.addEventListener(Event.ENTER_FRAME, animer);
    }
}
```

Nous pouvons constater ici qu'un test est réaliser afin de vérifier que l'élément `_objet` est bien instancié, et donc différent de la valeur **`null`**. Si ce n'est pas le cas, nous ne pouvons pas démarrer l'animation et choisissons de renvoyer une exception. Dans cette situation, l'erreur est détectée immédiatement et est bien de type synchrone. Dans la classe parente, une instruction **`try...catch`** permettra de gérer l'erreur.
Si aucune erreur n'est détectée, l'animation peut être lancée en enregistrant un écouteur sur l'événement `Event.ENTER_FRAME`.

Je vous invite à présent à tester le bon fonctionnement de la classe `Animation`. Essayons par exemple d'animer la position d'un objet jusqu'à la position (100,100) :

Code : Actionscript

```
var animation:Animation = new Animation(unObjet);
animation.incrementer( { x: 100, y:100 } );
```

Nous voyons ici que le programme ne pose aucun problème lors de l'exécution.

Vous remarquerez le passage de la valeur **null** en paramètre de la méthode `incrementer()` génère bien une exception.

Penchons-nous maintenant sur l'animation en elle-même, traitée à l'intérieur de la fonction de rappel `animer()`. Voici le contenu de cette méthode :

Code : Actionscript

```
private function animer(event:Event):void
{
    for (var p:String in _vars) {
        try {
            _objet[p]++;
            if (_objet[p] >= _vars[p]) {
                _objet.removeEventListener(Event.ENTER_FRAME,
animer);
            }
        }catch (e:Error) {
            _objet.dispatchEvent(new ErrorEvent("AnimErrorEvent"));
        }
    }
}
```

Concernant le fonctionnement, la boucle **for...in** permet de récupérer chacune des propriétés de l'objet `_vars`, sous forme de chaînes de caractères. À l'aide de ces dernières nous avons alors accès aux propriétés correspondantes de l'objet d'affichage, afin de pouvoir les incrémenter. Toutefois, il pourrait arriver que les propriétés renseignées ne soient pas de type numérique, ou n'existent tout simplement pas. Il serait alors avisé de renvoyer une erreur informant l'utilisateur qu'il tente d'accéder à une propriété inexistante, ou ne pouvant être incrémentée.

Pour réaliser ceci, commençons par isoler le code d'accès à la propriété en question. En effet, c'est cette instruction qui pourrait renvoyer une erreur, et qui permettra de déterminer si une erreur doit être renvoyée. Si l'erreur est avérée, celle-ci doit être renvoyée à l'utilisateur sous forme d'erreur asynchrone.

Il s'agit bien d'une exception qui est renvoyée en tant qu'erreur asynchrone à la classe de niveau supérieur. En effet, si cette erreur est synchrone à l'intérieur de la méthode `animer()`, celle-ci est néanmoins asynchrone par rapport à l'appel de la méthode `incrementer()` réalisé par l'utilisateur. Dites-vous bien qu'entre l'appel de cette dernière et l'apparition de l'erreur, il s'est écoulé un certain temps puisque nous sommes dans la fonction d'écouteur de l'événement `Event.ENTER_FRAME`.

Prenons l'exemple suivant afin de provoquer l'apparition d'une erreur :

Code : Actionscript

```
var animation:Animation = new Animation(objet);
animation.incrementer( { a:100, b:100 } );
```

L'apparition de l'erreur asynchrone a bien lieu :

Code : Console

```
Error #2044: AnimErrorEvent non pris en charge : text=
```

Vous remarquerez qu'en diminuant la cadence d'animation du projet, il est possible de percevoir très nettement le retard de génération de l'erreur asynchrone. Essayez par exemple de donner la valeur `1` à la propriété `frameRate` de l'objet `stage`. Vous verrez alors l'erreur n'apparaître qu'au bout d'une seconde dans la console de sortie.

Intérêts des erreurs

Les erreurs d'exécution ont principalement deux utilités.
D'une part, une erreur peut servir à informer le programmeur ou utilisateur d'une classe qu'il en fait un usage erroné. C'est notamment le cas lors de la conception de classes abstraites, comme nous l'avons vu, ou encore lorsqu'on souhaite avertir qu'un des paramètres d'une méthode n'est pas valide. Dans ce cas, le programmeur va revoir presque systématiquement son code afin de réparer l'erreur en question.
D'autre part, l'erreur peut être persistante et non réparable par le programmeur. Ce dernier peut alors contourner le problème afin de pouvoir poursuivre au mieux l'exécution du programme. C'est le genre de situation que nous rencontrerons dans la partie suivante, lorsque nous tenterons d'accéder à des données externes à l'application.

Dans tous les cas, la meilleure chose à faire est de *ne pas traiter l'erreur vous-mêmes*, mais de laisser la possibilité à l'utilisateur de la classe de gérer les erreurs comme il l'entend.

En résumé

- La **gestion des erreurs** consiste à ajouter du code spécifique au traitement des **erreurs d'exécution**.
- Les **erreurs synchrones** sont renvoyées immédiatement lors de l'exécution de l'instruction erronée.
- Les **erreurs asynchrones** correspondent aux erreurs ne pouvant pas être générées dans l'immédiat.
- L'instruction `try...catch` permet d'isoler un code erroné renvoyant une erreur synchrone ou **exception**.
- Une erreur asynchrone est représentée par la classe `ErrorEvent` ou une de ses sous-classes et peut être géré comme n'importe quel type d'événements.

~~Ces deux premières parties sont très théoriques, mais elles posent les bases pour la suite du cours, qui sera plus orientée sur la pratique, en commençant par l'affichage abordé dans la prochaine partie.~~

Nous espérons que ce cours vous a plu, et que vous souhaitez en savoir davantage ! Si vous le souhaitez, vous pouvez envoyer vos remarques et encouragements à AlphaDelta et à Guillaume. pour décupler notre motivation ! 

À bientôt pour la suite de l'aventure...

AlphaDelta et Guillaume.