Fascicule ActionScript

Les bases d’ActionScript

Sommaire

Sommaire 2
Coordonnées 3
Avant propos 4
1. Programmation séquentielle et programmation orientée objet 4
2. Quelques termes à connaître 5
Les bases d'ActionScript 6
1. Les variables 6
2. Les variables en programmation orientée objet 8
3. Les constantes 10
4. Les tableaux 10
5. Les structures conditionnelles 12
6. L'opérateur ternaire 13
7. La structure switch() 14
8. Les itérations : boucles for() 14
9. les fonctions 15
La gestion des évènements 17
1. Présentation 17
2. Le choix de l'évènement 18
3. Utilisation du clavier 19
4. La temporisation d'une action avec l'évènement ENTER_FRAME 20
Contrôler une occurrence 21
1. La méthode addChild() 21
2. Les encres 22
3. Les filtres 22
4. La couleur d'une occurrence 22
5. Rendre une occurrence mobile 23
6. Déplacer la tête de lecture du scénario 24
7. La classe *Tween* 25
Pour aller plus loin 27
1. Le chargement de médias sous forme de fichiers externes 27
2. Un exemple de classe 28

Coordonnées

Nicolas Lefèvre
Bureau SH 211
MRSH - Université de Caen Basse-Normandie
14032 Caen Cedex
Tel. 0231566238
Mail. nicolas.lefevre@unicaen.fr
Web. http://nlefevre.hbomb.fr

Avant propos

1. Programmation séquentielle et programmation orientée objet

Le langage ActionScript autorise les deux modes de programmation. Selon la taille du projet l'un ou l'autre des modes est préférable. En effet, si le projet est relativement petit, la programmation séquentielle ou, autrement appelée programmation fonctionnelle, est sans doute un bon choix. Pour les projets de grande envergures, il convient d'utiliser la logique de programmation orientée objet.

Ces deux modes de programmation sont assez différents. Dans le cas de la programmation orientée objet, nous allons développer une application d'un point de vue entité. Chaque élément sera, en faite, issu d'une structure disposant de propriétés et de fonctions associées. Ainsi, si deux entités sont issue de deux structures (appelées classes) différentes, ces entités n'auront pas les mêmes propriétés ni les mêmes opérateurs. De même, une autre différence avec la programmation fonctionnelle reste le fait qu'une entité à un comportement dans le temps qui peut se voir modifier par lui-même.

La programmation fonctionnelle ne prend pas en compte la structure de classe. Nous décrivons des fonctions générales qui produisent des résultats. Ces fonctions sont appelées à tout moment les unes après les autres. On parle alors de programmation séquentielle car nous appelons les fonctions les unes à la suite des autres.

La programmation orientée objet est plus complexe à maîtriser mais devient naturelle dès que sa philosophie est acquise. La programmation fonctionnelle est bien plus simple a mettre en œuvre.

2. Quelques termes à connaître

Instance : Lorsque nous utilisons un symbole dans une scène, nous utilisons une instance, ou occurrence. En ActionScript, le terme instance est un terme lié au langage orienté objet. Une instance de classe est objet créé par le mot clé new. Par exemple, pour la classe « personne », nous utilisons la ligne de code suivante :

```
var nouvellePersonne :new Personne() ;
```

Propriété : Une instance dispose de propriétés ; Par exemple, une instance de la classe « personne » dispose d'une propriété nom. Nous pouvons accéder a ce nom par la commande « . » comme suit :

```
nouvellePersonne.nom ;
```

Nous pouvons contrôler la position d'un objet en se referant à ses propriétés de position horizontale et verticale.

Méthode ou fonction : Dans un script, une fonction regroupe un ensemble de commande souvent utilisé. Nous pouvons faire référence aux fonctions partout dans un programme ce qui évite la réécriture des commandes. Lorsqu'une fonction est écrite à l'intérieure d'une classe, alors nous ne l'appelons plus fonction, mais méthode.

Evènement : Le fait de survoler une zone précise de la fenêtre de visualisation ou de cliquer sur une occurrence constitue un évènement. Nous pouvons exécuter un ensemble de commande (une fonction autrement dit) pour chaque type d'évènement. Cela nous permet de rendre réactif un bouton aux clics de souris.

Les bases d'ActionScript

1. Les variables

Une variable est un espace mémoire que nous allouons pour stocker une valeur ou un ensemble de valeurs. Cette valeur peut être un nombre, qu'il soit entier ou flottant, un caractère, une chaîne de caractère etc.

a. Déclaration d'une variable

Pour déclarer une variable dans ActionScript, nous utilisons la syntaxe suivante :

var** nomVariable**;

La principale caractéristique d'une variable, avec son contenu, est son type. Il représente le type de valeur stocké. Pour définir un type précis lors de la déclaration de la variable, nous utilisons de nouveaux mots clés :

var** nomVariable**:type ;

Comme nous pouvons le voir, le point virgule est précédé d'un espace. Cet espace n'est pas obligatoire. Voici plusieurs exemples de déclaration de variable :

*var nombre:**Number**;*
*var chaineCaracteres:**String**;*
*var binaire:**Boolean**;*
*var tableau:**Array**;*

Essayez le plus possible de nommer vos variables intelligemment. Si vous les nommez « variable_1 », « variable_2 » etc. vous n'aurez aucune chance de connaître la fonction de ces variables si vous reprenez votre programme quelques mois après l'avoir tapé. La nomenclature actuelle est la suivante : Nous n'écrivons pas la première lettre en majuscule, puis, si notre nom est composé de plusieurs mots, nous écrivons la première lettre de chaque mot après le premier en majuscule ainsi nous avons :

*var **voiciLeNomDeLaVariable** ;*

Une variable sert à stocker une valeur. Pour initialiser une variable a une certaine valeur, nous utilisons la syntaxe suivante :

***var** nomVariable**:type=valeur**;*
***var** nombreEntier:**Number**=5;*
***var** nombreEntier_2:**Number** = 5;*

La différence de syntaxe entre la déclaration de variable « nombreEntier » et « nombreEntier_2 » n'a aucune importance, en effet, les espaces autour du caractère « = » sont facultatifs.

Nous typons les variables pour profiter de l'aide du compilateur lors de la publication d'une animation. En effet, si nous ne typons pas nos variables, l'application peut fonctionner mais produire des résultats que nous n'attendons pas. En typant les variables, la compilation provoquera peut être des erreurs qui, une fois corrigées, fera que l'application produit les résultats escomptés. La seconde raison est une raison d'organisation. En typant nos variables nous organisons le programme d'une meilleure manière et la reprise du code est plus facile. Au même titre que le nommage des variables, le typage des variables est très important.

b. L'affectation de valeurs

Nous venons de voir comment initialiser une variable a une certaine valeur. Bien souvent, a l'initialisation, nous ne saurons pas la valeur de la variable. Pour modifier la valeur d'une variable nous utilisons l'opérateur « = » de la manière suivante :

```
var nombreEntier:Number;
nombreEntier=5;
```

*c. Le type **

Dans de rares cas, nous aurons besoin de typer une variable avec deux ou plusieurs types. Nous utilisons pour cela le type * dont voici un exemple de déclaration :

```
var elementListeArtcile:*;
```

d. La portée d'une variable

Nous le reverrons par la suite, mais sachez que l'endroit de déclaration d'une variable a une très grande importance. En effet, en déclarant une variable dans une fonction, cette variable n'est visible que depuis cette fonction, en la plaçant dans une boucle *for*, la variable n'est visible que dans la boucle et n'est pas accessible ailleurs. En d'autres termes, si nous voulons pouvoir utiliser une variable dans tout le code ActionScript, nous devons la déclarer au début du programme en dehors de toute fonction.

2. Les variables en programmation orientée objet

La portée d'une variable s'applique de la même façon en programmation orientée objet : si la variable est déclarer en dehors de toute fonction, elle sera utilisable dans chacune d'entre elle, tandis que si la variable est déclarée dans une fonction, elle ne sera visible que dans celle-ci.

```
package {
	public class Example {
		private var _id:String;

		public function Example( ) {
			_id = "Example Class";
		}

		public function getId( ):String {
			return _id;
		}
	}
}
```

a. *Les mots clé public, private, static*

Lorsque nous déclarons une variable dans une classe nous devons les préfixer du mot var, mais dans certains cas, nous pouvons ajouter un spécificateur de contrôle d'accès (*public*, *private* ou *static*)

Les variables déclarées « *static* » sont des variables qui ne sont pas instanciés, mais pour laquelle nous aimerions avoir accès. Le mot clé *static*, permet l'utilisation de la programmation séquentielle dans la programmation objet.

```
package {
	public class MainCreaTexte {
		public static var nom:String;

		public function MainCreaTexte( ) {
			new CreaTexte();
		}
	}
}

package {
	public class CreaTexte {

		public function CreaTexte( ) {
			MainCreatTexte.nom=”Bobo”;
			Trace(MainCreaTexte.nom);
		}

	}
}
```

Nous voyons bien dans l'exemple que la classe CreaTexte() fait référence a une variable, mais sans avoir instancié la classe MainCreaTexte() (c'est-à-dire, sans avoir utilisé le mot clé « new »). Une variable statique appartient à la classe et non à l'instance ce qui fait qu'il n'existe qu'une seule valeur pour cette variable à un moment donné.

Concernant les variables *public* et *private* c'est un peu plus simple. Une variable déclarée *public* sera accessible depuis une autre classe en faisant directement référence. Une variable déclarée private ne sera accessible que depuis sa classe et non depuis une autre. Nous utiliserons dans ce cas des fonctions appelées « setters » et « getters ». Ces traductions correspondent en fait à des fonctions qui permettent d'initialiser des variables ainsi qu'obtenir leurs valeurs.

```
package {
	public class Counter {
		private var _count:uint;
		public function Counter( ) {
			_count = 0;
		}

		public function getCount( ):uint {
			return _count;
		}

		public function setCount(value:uint):void {
			_count=value;
		}}}}
```

La classe Counter() ne nous donne pas accès à la variable « _count », du moins, pas directement a cause du mot clé « *private* » présent a la déclaration de la variable. Nous utilisons la fonction setCount(), qui fait office de « setters » pour appliquer une valeur a la variable d'instance. D'autre part, pour obtenir la valeur de cette variable, nous utilisons la fonction getCount(). Voici un exemple d'utilisation de la classe Counter :

```
package {
	public class ReadCounter {
		public var compteur:Counter;

		public function ReadCounter( ) {
			compteur = new Counter();
			trace(compteur._count);
			trace(compteur.getCount());
		}

	}
}
```

Nous avons deux lignes qui permettent d'afficher la valeur de la variable _count. Avec l'utilisation d'une variable *private* la première ligne ***trace(compteur._count)*** provoquera une erreur d'accession, il faudra en effet utiliser la seconde ligne, ***trace (compteur.getCount())***, qui est valide.

En utilisant les variables publiques, nous n'avons pas à nous soucier des problèmes d'accession, nous faisons directement référence a la variable comme à la ligne ***trace(compteur._count);***

3. Les constantes

Une information est parfois stockée dans une variable sans qu'il soit necessaire de pourvoir la modifier. Nous appelons ce genre de variable « qui ne varie plus du tout) des contstantes. Elles se déclarent avec le mot clé const, ainsi :

```
const var pi :Number = 3.1415 ;
const var TVA55 :Number = 0.055 ;
```

4. Les tableaux

a. *Tableau à 1 dimensions*

Les tableaux sont des structures accueillant un grand nombre de données. Une seule déclaration de tableau peut nous permettre de structurer un ensemble de données. La déclaration d'un tableau se fait toujours grâce au mot clé var, cependant, la variable aura un type précis de données, le type *Array* :

```
var demiSemaine:Array = new Array("lundi", "mardi", "mercredi");
var tableauVide:Array=new Array();
```

Il faut bien noter plusieurs choses ici. La première concerne le type à utiliser et la seconde concerne l'affectation des données. Nous utilisons le mot clé *new* qui signifie que nous instancions la classe *array*. Il existe une seconde méthode de déclaration de tableau beaucoup plus généraliste (ie. Que nous retrouvons dans bon nombre de langages de programmation) :

```
var demiSemaine:Array = ["lundi", "mardi", "mercredi"];
```

Un tableau peut ne pas stocker qu'un type de variable. Chacun de ses espaces est indépendant, ainsi, la déclaration suivante est valide :

```
var data:Array = ["a", 2, true, new Object()];
```

Attention toutefois, une telle structure est très difficile a manipuler. De par son organisation, l'accession aux valeurs est délicate. Il vaut mieux utiliser un tableau contenant un seul type de donnée.

b. *Tableau à deux dimensions, l'accession aux valeurs et la modification des tableaux*

Jusqu'à maintenant, nous n'avons déclaré qu'une liste d'éléments. Un tableau se compose généralement d'au moins deux lignes et de deux colonnes. Voici la déclaration d'un tableau à deux dimensions :

```
var colors:Array = ["maroon", "beige", "blue", "gray"];
var years:Array = [1997, 2000, 1985, 1983];
var makes:Array = ["Honda", "Chrysler", "Mercedes", "Fiat"];
var car:Array = [colors,years,makes];
```

Le tableau nommé « car » contient un ensemble de données déclarées par les variables *colors*, *years* et *makes*. Nous pouvons accéder à une des données du tableau par l'écriture suivante :

```
car[0][0]; -> maroon
car[1][3]; -> 1985
car[2][1]; -> Chrysler
```

Lors de l'utilisation d'un tableau à une seule dimension, nous utilisons la syntaxe suivante pour accéder aux valeurs (indice est un chiffre) :

```
tableau_1_dimension[indice] ;
```

De même, pour un tableau à trois dimensions[1], nous utilisons l'écriture suivante :

```
tableau_3_dimensions[indice_dim_1][indice_dim_2][indice_dim_3] ;
```

L'accession aux valeurs nous permet aussi de modifier le contenu du tableau, ainsi, si nous reprenons notre tableau de demi semaine, nous pouvons lui affecter d'autres valeurs en couplant l'écriture avec le mot clé d'affectation « = » :

```
demiSemaine[0] = "Dimanche";
```

Cet exemple supprimera la donnée « lundi » qui était présente à l'indice 0 du tableau d'origine. Pour insérer des données nous avons les possibilités suivantes :

```
demiSemaine.push("Jeudi"); -> ajoute jeudi à la fin du tableau.
demiSemaine.unshift("dimanche"); -> Insère au début du tableau.
demiSemaine.splice(2,0, "Jeudi") ; -> Insère à un endroit dans le
  tableau.
```

[1] Généralement, nous n'allons jamais plus loin que deux dimensions, mise a part des cas spéciaux, le tableau à trois dimensions est difficilement utilisable.

Pour la fonction splice, le premier paramètre indique l'indice d'insertion dans le tableau, tandis que le second indique le nombre de données à supprimer (En indiquant 0, aucun élément n'est écrasé).

Il est possible de supprimer un élément d'un tableau, pour cela nous avons une fonction globale qui peut fonctionner dans n'importe quel cas de suppression :

```
demiSemaine.splice(i, 1) ; -> i est l'indice de l'item à supprimer.
demiSemaine.splice(0,1) ; -> supprime le premier élément.
demiSemaine.shift() ;-> supprime le premier élément.
demiSemaine.splice(demiSemaine.length-1,1) ; -> supprime le dernier
  élément.
demiSemaine.pop() ; -> supprime le dernier élément.
```

La quatrième ligne de code nous donne accès a un nouveau mot clé, le mot clé « length » qui renvoi la longueur d'un tableau. Attention, pour accéder au dernier élément, nous utilisons le mot clé « length » - 1 car les indices de tableaux commencent à 0.

5. Les structures conditionnelles

L'interactivité dans Flash est très souvent gérée par l'évaluation de certaines conditions. La plupart des scripts que nous rédigerons contiendront des structures conditionnelles.

a. La structure conditionnelle if() else()

Voici comment écrire une condition if() else(). Il existe deux possibilité, l'une avec accolade, et l'autre sans :

```
if(condition) resultat_vrai
else resultat_faux


if(condition) {
    Resultats_vrais
}
else {
    resultats_faux
}
```

Dans le premier cas, nous n'utilisons pas d'accolade, ce qui signifie que le résultat de la structure conditionnelle n'est qu'une simple ligne de code. Dans le second cas, les accolades permettent de produire une séquence de code dans la structure. Voici l'exemple suivant : « S'il n'y a plus que 2 joueurs, alors nous sommes en finale, sinon, nous n'y somme pas encore ».

```
if(nombre_joueur==2) trace("C'est la finale");
else trace("Ce n'est pas encore la finale");
```

Si nous devions faire plusieurs commandes dans la même structure, nous utiliserions les accolades de la manière suivante :

```
if(nombre_joueur==2) {
    trace("C'est la finale");
    trace(joueur_1.nom);
    trace(joueur_2.nom);
}
else  {
    trace("Ce n'est pas encore la finale");
    trace("Trop de joueurs pour afficher les noms");
}
```

Comme nous pouvons le voir, la condition n'est qu'un test entre deux valeurs. En fonction du résultat de ce test, nous exécutons une séquence ou une autre. Les tests ne sont pas forcement que des tests d'égalités, les opérateurs disponibles sont :

- Le test de superiorité noté >
- Le test d'infériorité noté <
- Le test de différence noté !=
- Le test d'égalité noté == **(2 fois le signe égal)**[2]
- Il existe aussi les test « inférieur ou égal » noté <= (respectivement >= pour superieur ou égal)

b. Les conditions multiples

Il peut arriver que nous devions produire plusieurs tests. Nous avons deux solutions là encore, soit nous imbriquons plusieurs structures if() else() les unes dans les autres, soit nous utilisons les deux opérateurs logiques « ou » noté || ou « et » noté &&. Voici un exemple :

```
if(nombre_joueur==2) {
    if (joueur_1.nation=="France" || joueur_2.nation=="France"){
        trace("C'est la finale avec un joueur français!");
    }
    else trace("C'est la finale, mais il n'y a pas de français.");
}
else  {
    trace("Ce n'est pas encore la finale");
    trace("Trop de joueurs pour afficher les noms");
}
```

6. L'opérateur ternaire

Il existe une autre façon d'écrire une structure conditionnelle. Elle est beaucoup plus difficile à assimiler de part son écriture, mais elle devient plus simple lorsqu'elle est assimilée :

```
valeur=test ? valeur si vrai : valeur si faux ;
message = nombre_joueur==2 ? "C'est la finale":"Ce n'est pas la
   finale" ;
trace(message);
```

[2] En effet, un seul signe égal est une affectation de valeur à une variable, donc le résultat du test avec un seul signe égal rend de très mauvais résultats !

7. La structure switch()

Si nous devons écrire un ensemble de condition les unes a la suite des autres, la structure switch() peut nous aider, voici un exemple d'utilisation :

```
switch (joueur.nation) {
	case "France" :
		trace("capitale : Paris") ;
		break ;
	case "Italie" :
		trace("capitale : Rome") ;
		break ;
	case "Espagne" :
		trace("capitale : Madrid") ;
		break ;
	default :
		trace("Nation inconnue dans le prog.") ;
}
```

Dans cette structure, la nation du joueur va être tester parmi toutes les nations disponibles (France, Italie ou Espagne). Dès qu'une nation correspond, nous écrivons la capitale. Dans le cas ou la nation du joueur n'est pas disponible dans nos tests, nous écrivons qu'il n'y a pas de capitale associée à la nation du joueur. Si une condition est valide, nous stoppons l'exécution de la structure switch() par le mot clé *break*.

8. Les itérations : boucles for()

La boucle for sert à exécuter plusieurs fois le même segment de code. L'ecriture peut être un peu délicate au début, mais elle reste très logique :

```
for(définition de la boucle){code à executer}
```

La plus compliqué reste de définir la boucle. Nous avons besoin de trois composantes : Une variable initialisée (un nombre entier), une condition d'arrêt de boucle, et un pas d'augmentation pour la variable. Imaginons que nous voulions compter de 0 à 100 par pas de 1, nous écrivons la ligne suivante :

```
for (var variable:Number=0 ;variable<=100 ; variable = variable+1){
	trace (variable);
}
```

Voici un autre exemple d'addition des chiffres de 1 à 100 (Notez par la même occasion la porté de la variable résultat : En la déclarant en dehors de la boucle for, nous pouvons l'utiliser à la sortie de cette boucle ou à l'intérieur) :

```
var resultat:Number=0 ;
for (var variable:Number=0 ;variable<=100 ; variable = variable+1){
      resultat = resultat + variable
}
trace (resultat);
```

Nous pouvons, grâce à la boucle for, parcourir un tableau d'éléments[3] :

```
for (var variable:Number=0 ;variable<=(tableau.length-1);variable++){
      trace (tableau[variable]);
}
```

9. les fonctions

a. *La déclaration de fonctions*

Une fonction est un ensemble de commande regroupée sous un nom. Dans ActionScript, nous avons trois types de fonctions :

- Les fonctions simples
- Les fonctions avec paramètres
- Les fonctions de rappel qui fonctionnent avec les gestionnaires d'évènements

Voici comment déclarer une fonction en ActionScript :

```
function nom_fonction(paramètres){
      Commandes a exécuter
}
```

Voici deux exemples de déclaration de fonction dans la première, il n'y a pas besoin de paramètres (Lorsque plusieurs paramètres sont nécessaires, nous utilisons le séparateur virgule) :

```
function afficheBonjour(){
 trace("Bonjour ! ");
}

function afficheMessage(message:String, repetition:Number){
 for(var i:Number=0 ; i<repetition ; i++)
       trace(message) ;
}
```

La déclaration d'une fonction est une chose, l'appel à la fonction en est une autre. Pour appeler une fonction nous devons juste écrire le nom de la fonction, ainsi :

afficheBonjour() ; -> affichera Bonjour !
afficheMessage("Au revoir", 5) ; -> affichera 5 fois « au revoir »

[3] Le segment variable++ est une écriture abrégée de variable = variable + 1

b. Les fonctions récursives

Une fonction peut s'appeler elle-même. L'exemple le plus connu est celui de la fonction exponentielle. Pour rappel la fonction exponentielle d'un chiffre « i » donne comme résultat la multiplication entre eux des chiffres de 1 à « i ». Voici la fonction exponentielle en ActionScript :

```
function exponentielle (i:Number){
 if (i<=1) return 1 ;
 else trace(i*exponentielle(i-1)) ;
}
```

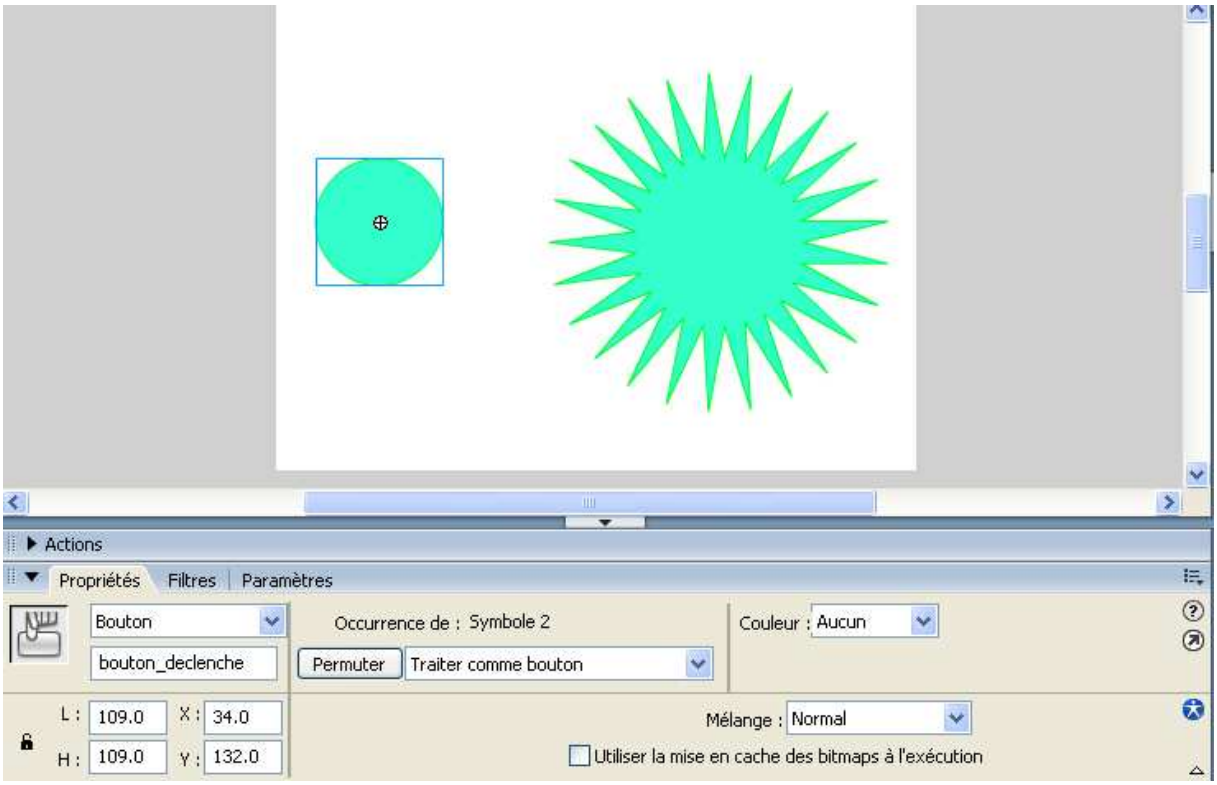
Evidement, puisque nous pouvons appeler une fonction au sein de même de cette dernière, nous avons la possibilité de faire appel à une fonction extérieure.

La gestion des évènements

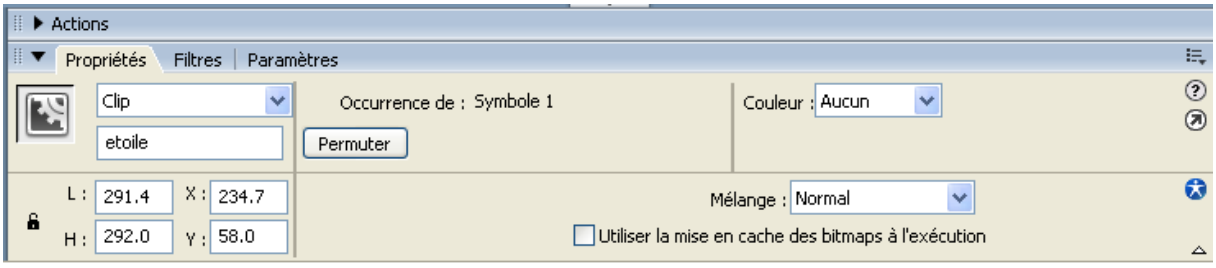
1. Présentation

Nous arrivons aux choses intéressantes avec ActionScript. Nous allons voir comment faire réagir les entités aux interactions de l'utilisateur. Le gestionnaire d'évènements définit la relation qui existe entre une occurrence de symbole et une fonction contenant le code à exécuter lorsque l'évènement survient. Nous verrons qu'il existe bon nombre d'évènement, dont le plus connu, le clic de souris.

Il n'existe qu'une seule méthode pour gérer l'interactivité d'une animation, elle est quelques peu complexe mais on fini par s'y faire. Par la suite, nous utiliserons deux exemples d'occurrences : l'une étant une étoile l'autre un bouton. L'étoile est nommé « etoile » tandis que le bouton est nommé « bouton_declenche ».



Nous nommons les occurrences a partir de l'inspecteur de propriété du symbole.



La première chose que nous allons faire est que nous allons faire tourner l'étoile sur elle-même lorsque nous cliquons sur le bouton. Pour cela nous devons déclarer une fonction spéciale de par ses paramètres pour que le gestionnaire d'évènements puisse la prendre en compte :

```
function tourne_etoile(evt:MouseEvent){
   etoile.rotation+=5 ;
}
```

Comme nous le voyons, la fonction possède forcement un paramètre de la classe MouseEvent. L'oubli de ce paramètre entraîne une erreur lors de l'exécution du programme. Il s'agit d'une fonction de rappel.

Notre fonction étant établi, nous devons déclarer un écouteur. Un écouteur va attendre qu'un évènement se produise pour activer une fonction. Nous allons placer un écouteur de clic de souris sur le bouton nommé « bouton_déclencheur ». La déclaration est la suivante :

```
nom_occurence.addEventListener(évènement,fonction);
```

Si nous voulons que notre bouton écoute l'évènement « clic de souris » et qu'il active la fonction « tourne_etoile » le cas échéant, nous écrivons :

```
bouton_declencheur.addEventListener(MouseEvent.MOUSE_DOWN,tourne_etoile);
```

Voila, a partir de ce moment, lorsque le bouton de la souris est appuyé sur le bouton « bouton_declencheur », l'étoile tourne de 5 degrés.

2. Le choix de l'évènement

Comme nous l'avons vu en introduction de cette partie, il existe bon nombre d'évènements dont voici la liste principale :

- MouseEvent.CLICK
- MouseEvent.DOUBLE_CLICK
- MouseEvent.MOUSE_DOWN
- MouseEvent.MOUSE_MOVE
- MouseEvent.MOUSE_OUT
- MouseEvent.MOUSE_OVER
- MouseEvent.MOUSE_UP
- MouseEvent.MOUSE_WHEEL
- MouseEvent.ROLL_OUT
- MouseEvent.ROLL_OVER

Reprenons notre fonction permettant de tourner l'étoile. Nous pouvons nous servir du paramètre de la classe MouseEvent pour faire tourner non pas une entité définie, mais plusieurs. Voici le code modifié de la fonction :

```
function tourne_objet (evt:MouseEvent){
   evt.currentTarget.rotation+=5 ;
}

etoile.addEventListener(MouseEvent.MOUSE_DOWN,tourne_objet);
autre_entite.addEventListener(MouseEvent.MOUSE_DOWN,tourne_objet);
```

Avec cette écriture, nous disposons d'une seule fonction, mais celle-ci est applicable à plusieurs entités. Il faut pour cela qu'elle gère un évènement qui aiguillera sur la fonction « tourne_objet ». Dans l'exemple, nous avons deux entités, l'une étant notre étoile, et l'autre étant « autre_entite »[4].

Nous pouvons aller plus loin avec le paramètre de la classe MouseEvent. En effet, il ne stocke pas que le nom de l'occurrence, il stocke aussi d'autres valeurs dont le type d'évènements, ainsi, nous pouvons écrire :

```
function tourne_objet (evt:MouseEvent){
   if(evt.type=="MouseDown")
         evt.currentTarget.rotation+=5 ;
   else evt.currentTarget.rotation+=-5 ;
}

etoile.addEventListener(MouseEvent.MOUSE_DOWN,tourne_objet);
etoile.addEventListener(MouseEvent.MOUSE_UP,tourne_objet);
```

Dans le cas présent, si le clic est de type MouseDown, alors nous effectuons une rotation de 5 degrés, sinon, pour tout autre évènement, nous effectuons une rotation de -5 degrés. En attachant plusieurs écouteurs sur la même occurrence de symbole, nous pouvons gérés plusieurs évènements.

Du point de vue de l'interactivité, nous utilisons principalement l'évènement MOUSE_UP plutôt que MOUSE_DOWN. Lors de l'utilisation de MOUSE_DOWN, la fonction est appelé dès le clic, a la différence avec MOUSE_UP qui déclenche la fonction après que la souris soit relâchée, cela laisse le temps à l'utilisateur de voir qu'il s'est tromper lors d'un clic.

3. Utilisation du clavier

Nous pouvons utiliser le clavier plutôt que la souris. Pour cela, nous devons modifier notre fonction. Elle ne doit plus utiliser le paramètre de la classe MouseEvent, mais cette fois-ci, nous utilisons un paramètre de la classe KeyboardEvent, ainsi, nous avons :

```
function tourne_objet (evt:KeyboardEvent){
    if(evt.keyCode==37)
          evt.currentTarget.rotation+=5;
    if(evt.keyCode==39)
          evt.currentTarget.rotation+=-5;
}

etoile.addEventListener(KeyboardEvent.KEY_DOWN,tourne_objet);
```

Avec un tel code, lorsque l'utilisateur appuiera sur la flèche droite ou gauche du clavier, une rotation sera effectuée.

[4] Evidement, il faut que nos entités existent en tant qu'occurrence de symbole…

4. La temporisation d'une action avec l'évènement ENTER_FRAME

Jusqu'à présent, nous avons abordés les évènements liés à l'interaction avec l'utilisateur. Imaginons maintenant que nous voulions qu'une action s'exécute sans que l'utilisateur n'y soit pour quelque chose, mais en plus, que cette action se poursuive en continu. Nous utilisons pour cela l'évènement ENTER_FRAME de la classe Event :

```
function tourne_etoile(evt: Event){
    etoile.rotation+=5 ;
}

etoile.addEventListener(Event.ENTER_FRAME,tourne_etoile);
```

Le code ne doit pas s'arrêter là. En effet, si nous exécutons ce code, l'étoile se mettra à tourner et plus rien ne pourra l'arrêter. Nous devons programmer une fonction permettant l'arrêt de la rotation.

```
function arrete_etoile(evt :MouseEvent){
    etoile.removeEventListener(Event.ENTER_FRAME,tourne_etoile);
}

etoile.addEventListener(MouseEvent.MOUSE_UP,arrete_etoile) ;
```

Cela peut paraître bizarre d'ajouter un écouteur pour en arrêter un autre, toutefois, il n'y a aucun autre moyen. Par la suite, nous verrons d'autres types d'écouteurs un peu plus évolués. D'une manière générale, la déclaration d'écouteur se fait naturellement par la même fonction, l'évènement étant passé en paramètre. La syntaxe du gestionnaire n'est pas le plus compliqué, ce qu'il l'est c'est de mémoriser les principaux évènements.

Contrôler une occurrence

1. La méthode addChild()

Cette méthode constitue l'action élémentaire pour placer dynamiquement un objet d'affichage ou un conteneur d'affichage sur la scène. Lorsque cette méthode est utilisée, l'occurrence est ajoutée à la liste d'affichage.

En d'autres termes, lorsque nous aurons besoin de placer un symbole de la bibliothèque sur la scène, nous utiliserons la méthode *addChild()*. Cette dernière n'a qu'un seul rôle, rendre visible un objet sur la scène. Voici un exemple d'utilisation (Nous la verrons dans un exemple plus concret dans la partie consacrée au chargement dynamique des images) :

```
var spriteEntete:Sprite=new Sprite() ;
addChild(spriteEntete);
```

Pour contrôler une occurrence, nous devons connaître ses principales propriétés.

- Les propriétés *x* et *y* servent à contrôler la position (exprimée en pixels) de l'occurrence. L'orientation peut être réglé par la propriété *rotation* exprimée en degrés.
- La propriété *visible* sert à afficher ou masquer l'occurrence exprimée par une valeur booléenne.
- Le réglage de la transparence est géré par la propriété *alpha* exprimée entre 0 et 1.
- Les propriétés *scaleX* et *scaleY* sont utilisées pour modifier l'échelle de l'occurrence exprimée entre 0 et 1 pour une réduction et au delà de 1 pour un agrandissement.
- Les dimensions sont déterminées par les propriétés *width* et *height* exprimées en pixels.

Pour modifier une de ces propriétés, c'est très simple, nous devons utiliser le nom de l'occurrence suivi du nom de la propriété séparé par un point. Cela nous permet d'affecter une valeur ou de connaître la valeur courante :

```
etoile.x ; -> donne la position courante de l'occurrence nommée étoile
etoile.x=50 ; -> positionne l'étoile à une position x de 50.
etoile.alpha=0.5 ; -> affiche l'étoile avec une transparence de 50%.
etoile.scaleX=étoile_2.scaleX+0.2; -> place la taille x de l'étoile à celle
de l'étoile 2 + 20%.

etoile.rotation=etoile_2.rotation=25 ; -> double affectation de valeur aux
deux étoiles.
```

2. Les encres

Les encres sont des propriétés permettant de modifier le mode de surimpression d'une occurrence par rapport au contenu en arrière plan. Il s'agit du paramètre *mélange* de l'inspecteur de propriété d'un symbole. Pour le modifier, nous écrivons :

```
nom_occurence.blendMode = BlendMode.encre ;
```

Il existe 14 encres utilisables dont voici la liste : ADD, ALPHA, DARKEN, DIFFRENCE, ERASE, HARDLIGHT, INVERT, LAYER, MULTIPLY, NORMAL, OVERLAY, SCREEN, SUBSTRACT. Par exemple, nous pouvons écrire :

```
etoile.blendMode = BlendMode.INVERT ;
```

3. Les filtres

Toujours dans l'inspecteur de propriétés, nous connaissons l'onglet filtres. Nous pouvons ajouter un filtre a une occurrence par ActionScript. Pour cela, nous allons avoir besoin d'une séquence de programmation assez particulière. Tout d'abord, nous n'appliquons pas un filtre directement a une occurrence. En fait, chaque occurrence possède une propriété *filters* qui est un tableau (vide à l'origine) qui contient chacun des filtres appliqués. Ensuite, nous devons importer des classes qui ne le sont pas par défaut. Pour cela nous utilisons le mot clé *import*. Voici un exemple :

```
import flash.filters.dropShadowFilter ;

var ombre:DropShadowFilter = new DropShadowFilter() ;
var listeFiltres :Array = new Array() ;
listeFiltres.push(ombre);
etoile.filters=listeFiltres;
```

Ainsi écrit, nous n'avons soumis aucune valeur pour l'ombre. En fait, chaque filtre dispose de ces propos paramètres, ils sont trop nombreux pour être expliqué ici. L'aide de Flash pourra nous fournir l'aide que nous souhaitons. La chose la plus importante reste qu'il ne faut pas oublier d'importer la classe du filtre que nous souhaitons utiliser.

4. La couleur d'une occurrence

Pour modifier la couleur d'une occurrence, nous utilisons aussi une séquence de programmation. Nous avons besoin d'une instance de la classe *ColorTrasnform*. Voici un exemple (Dans ce dernier, nous donnons une couleur verte à l'étoile ; referez-vous aux systèmes hexadécimaux de gestion des couleurs RVB) :

```
var nouvelleCouleur : ColorTransform = new ColorTransform() ;
nouvelleCouleur.color = 0x00ff00 ;
etoile.transform.colorTransform=nouvelleCouleur ;
```

5. Rendre une occurrence mobile

Pour pouvoir déplacer une occurrence (autrement dit, modifier ses propriétés *x* et *y*) à l'aide de la souris, nous utilisons la méthode *startDrag()* :

```
etoile.startDrag() ;
```

Nous pouvons allez un peu plus loin avec cette méthode de déplacement automatique. En effet, si nous lançons cette composition avec cette ligne de code placé dans la première frame, l'étoile suivra notre curseur. Nous pouvons imaginer que cette étoile est notre curseur, par conséquent, il faudrait pouvoir masquer le curseur de windows. Nous utilisons les lignes suivantes :

```
Mouse.hide() ; -> masque le curseur
Mouse.show() ; -> affiche le curseur
```

En utilisant un évènement nous pouvons contrôler l'activation du déplacement d'une occurrence. Toujours avec notre étoile, saisissons cette séquence de programmation :

```
etoile.addEventListenet(MouseEvent.MOUSE_DOWN,etoileMobile) ;
etoile.addEventListenet(MouseEvent.MOUSE_UP,etoileStatique) ;

function etoileMobile(evt :Event){
      étoile.startDrag() ;
}

function etoileStatique(evt :Event){
      stopDrag() ;
}
```

Dans cet exemple, nous actionnons le déplacement de l'étoile lorsque le bouton de la souris est appuyé, et nous arrêtons grâce à la méthode *stopDrag()* lorsque le bouton est relaché. Nous produisons un glisser déposer en ActionScript.

La méthode *startDrag()* peut prendre en paramètres plusieurs variables dont l'une, des plus intéressantes) concernent la contrainte de déplacement. Imaginons que notre étoile ne puisse se déplacer qu'horizontalement, nous devons définir une zone rectangulaire de déplacement de la manière suivante :

```
var zoneLimite :Rectangle = new Rectangle(50,200,250,0);
   -> Cela nous donne un rectangle dont le coin haut gauche est situé
   à la coordonnée 50,200, et d'une longueur de 250 pour une hauteur
   de 0.
etoile.startDrag(false,zoneLimite) ;
```

Nous pouvons tester la collision entre deux occurrences. Pour cela, nous utilisons la méthode *hitTestObject()*. Cette méthode d'instance prend en paramètre une seconde occurrence et retourne la valeur booléenne *true* s'il y a collision, ou *false* le cas contraire :

```
etoile.hitTestObject(etoile_2) ;
```

6. Déplacer la tête de lecture du scénario

Pour placer la tête de lecture du scénario principal, nous utilisons la fonction *gotoAndStop()*. Elle prend en paramètre un nombre qui représente la frame sur laquelle la tête de lecture doit être placée :

```
gotoAndStop(1) ; -> se rend a la première image du scénario
```

Cette fonction permet, par exemple, de modifier l'affichage en réaction à un évènement, nous plaçons la tête de lecture du scénario a un autre endroit contenant des entités graphiques complément différentes. Il existe d'autres méthodes ayant un effet sur la tête de lecture, en voici quelques unes :

```
stop() ; -> arrête la lecture.
play() ; -> relance la lecture après qu'elle soit interrompue avec la
   méthode stop ou gotoAndStop.
gotoAndPlay(indiceImage) ; -> déplace la tête de lecture du scénario et
   relance la lecture.
nextFrame() ; -> déplace la tête de lecture sur l'image suivante.
prevFrame() ; > déplace la tête de lecture sur l'image précédente.
```

Nous utilisons les mêmes fonctions pour déplacer les têtes de lectures des clips d'animations. Cette fois-ci, toutefois, nous devons préfixer le nom de la fonction du nom de l'occurrence. Si l'occurrence nommée « etoile » est un clip d'animation, nous pouvons écrire :

```
etoile.nextFrame();-> fait passer la tête de lecture du clip étoile sur
   l'image suivante.
etoile.gotoAndStop(20);-> place le clip à l'image 20 de son animation.
```

7. La classe *Tween*

La classe *Tween*, interpolation en anglais, permet une interpolation de mouvement. Cette classe nécessite l'import de deux classes *easing* et *transitions*. Sans importer ces deux classes, nous ne pourrons pas faire appel à la classe *Tween*. Par ailleurs, les propriétés de position, orientation, échelle etc. disponibles sur une occurrence de symbole seront utilisables dans la classe *Tween*.

```
import fl.transitions.easing.* ;
import fl.transitions.*;
```

Découvrons à présent l'unique ligne d'instruction nécessaire au déroulement d'un menu :

```
new Tween(nomOccurence, "y",Regular.easeOut,20,180,2,true) ;
```

Une ligne comme cette dernière peut paraître difficile a comprendre, voici l'explication de chacun de ses paramètres :

- nomOccurence n'est pas compliqué à comprendre. Nous devons bien évidement spécifier l'occurrence qui devra faire un mouvement.
- *y* est le nom de la propriété concernée par le mouvement. Nous pouvons utiliser les propriétés *x, y,scaleX,scaleY,width,height,alpha* et *rotation*.
- Regular.easeOut est un mode d'animation pour obtenir un effet. Nous reviendrons sur ce paramètre.
- 20 : Ce paramètre est exprimé pour la propriété passée en paramètre, ici *y*. Le point d'ancrage de l'occurrence se placera à une position de 20 pixels en hauteur. Il s'agit de la valeur de départ de l'interpolation.
- 180 : Valeur d'arrivée de l'interpolation exprimée pour la propriété passée en paramètre. Il s'agit également du point d'ancrage.
- 2 : Durée de l'interpolation en secondes.
- *true* : Paramètre servant à préciser que le dernier paramètre est exprimé en secondes et non en images.

Couplé avec un système de gestionnaire de souris, il est très facile de concevoir une interactivité sur un menu par exemple. Voyons un peu plus en détail ce que nous permet le troisième paramètre d'une instance de la classe *Tween*.

Il existe plusieurs modes d'animation, chacun d'entre eux produit des résultats différents. Le plus simple est de tester les combinaisons disponibles, à la place de *Regular*, nous pouvons écrire :

- Back
- Bounce
- Elastic
- Strong

A cela s'ajoute trois paramètres qui définissent l'application de l'effet en début ou en fin de trajectoire. Attention toutefois, les résultats peuvent être très différents selon les options utilisées. En plus de ces paramètres, il convient de bien choisir son temps d'interpolation :

- easeIn
- easeOut
- easeInOut

Pour terminer, nous allons voir les propriétés de la classe *Tween* ainsi que les méthodes qu'elle nous propose pour contrôler plus efficacement nos interpolations. Voici les propriétés :

```
bouge = new Tween( nomOccurence, "y", Regular.easeOut ,20,180,2,true) ;
```

- *begin* : Valeur de départ de l'interpolation, ici 20.
- *duration* : Durée de l'interpolation, ici 2.
- *finish* : valeur de fin de l'interpolation, ici 180.
- *FPS* : Frame par seconde, ou IPS pour image par seconde.
- *func* : fonction d'accélération utilisée
- *isPlaying* permet de savoir si l'interpolation est en cours.
- *looping* permet de préciser si l'interpolation doit être lue en boucle.
- *obj* : nom de l'objet, ici nomOccurence.
- *position* permet de connaître la valeur actuelle du paramètre animé, ici une valeur comprise entre 20 et 180.
- *prop* permet d'obtenir la propriété soumise a l'interpolation, ici, *y*.
- *time* permet de connaître le temps écoulé depuis le début de l'interpolation
- *useSeconds* permet de savoir si l'interpolation est exprimée en secondes.

Voici maintenant les méthodes de la classe *Tween*.

- *continueTo()* permet de poursuivre l'interpolation vers une nouvelle valeur. Cette fonction ne contient pas de valeur par défaut car elle défini une nouvelle trajectoire a partir de la position courante de l'objet. Seules les valeurs de départ et d'arrivée doivent être précisés.
- *fforward()* termine l'interpolation sans exécuter les interpolations intermédiaires.
- *nextFrame()* permet d'afficher l'image suivante lorsque l'interpolation est arrêtée.
- *prevFrame()* permet d'afficher l'image précédente lorsque l'interpolation est arrêtée.
- *resume()* relance l'interpolation lorsqu'elle à été mise en pause à l'aide de la fonction *stop()*.
- *rewind()* réaffecte la valeur de départ à l'occurrence interpolée.
- *start()* lit une interpolation depuis sa valeur de départ.
- *stop()* permet d'interrompre l'interpolation.
- *yoyo()* permet d'effectuer une interpolation dans le sens inverse.

Pour aller plus loin

1. Le chargement de médias sous forme de fichiers externes

a. Les images

Nous pouvons charger des medias dynamiquement a partir d'ActionScript, cela permet d'alléger le poids des fichiers SWF et leur mise a jour est facilitée. Commençons par voir les classes nécessaires à l'import de données. Nous aurons besoin de la classe *Loader* et de la classe *URLRequest*. Nous commencons par créer deux instances de classes :

```
var chargeur :Loader() ;
var adresseImage :URLRequest=new URLRequest("image.jpg");
```

La première classe va nous servir de conteneur pour l'image dont l'adresse est encapsulée par la seconde. Il nous faut ensuite ajouter l'image à la liste d'affichage :

```
chargeur.load(adresseImage) ;
addChild(chargeur) ;
```

Nous terminons l'intégration en positionnant notre image (par son coin supérieur gauche) à l'endroit désiré sur la scène :

```
chargeur.x=150 ;
chargeur.y=25 ;
```

Dans cet exemple nous chargions l'image directement dans la scène. Nous pouvons charger une image dans une instance de la classe *Sprite*, ce qui nous permet de modifier le point d'alignement de l'image. Dans ce cas, nous instancions la classe *Sprite* et nous ajoutons cette instance à la liste d'affichage :

```
var cadre :Sprite = new Sprite() ;
addchild(cadre);
cadre.x=250;
cadre.y=112;

var chargeur :Loader() ;
var adresseImage :URLRequest=new URLRequest("image.jpg");
chargeur.load(adresseImage) ;
chargeur.x=-105 ; -> pour une image d'une taille de 210 en x
chargeur.y=-105 ; -> pour une image d'une taille de 210 en y

cadre.addChild(chargeur) ;
```

b. *Les sons*

Pour les sons, nous procédons de la même manière sauf que cette fois ci, nous instancions la classe *Sound*. Voici un exemple de lecture d'un son chargé dynamiquement :

```
    var adresseSon :URLRequest=new URLRequest("son.mp3");
    var enceinte = new Sound(adresseSon) ;
    enceinte.play();
    //enceinte.play(0,4); -> précise le déclanchement de 4 lectures dès le
début de la publication.
    //enceinte.stop() ; -> arrête le son
```

2. Un exemple de classe

ActionScript autorisant la programmation orientée objet, ce langage nous donne la possibilité d'écrire nos propres classe. Cela sert principalement lors de la réalisation de gros projets. Imaginons une classe *Personne()* qui pourrait servir au traitement d'une liste dans une administration, ou autre… Chaque personne dispose d'un nom, d'un prénom et d'une date de naissance. Voici la déclaration simplifiée de la classe, suivi de quelques méthodes pouvant être utile :

```
package{
  public class Personne {
    private var nom :String ;
    private var prenom :String ;
    private var anneeNaissance :Number ;

    public function Personne(_nom :String,_prenom :String,_annee : String){
      this.nom=_nom;
      this.prenom=_prenom;
      this.anneeNaissance=_annee;
    }

    public boolean function egalite(_pers:Personne){
      return(this.nom==_pers.nom&&this.prenom=_pers;prenom&&this.anneeNaissan
        ce=_pers.anneeNaissance);
    }

    public Number calculAge(anneeCourante:Number){
      return (anneeCourante-this.anneeNaissance);
    }
  }
}
```

La fonction portant le même nom que la classe est appelée *constructeur*. On l'appelle ainsi car lorsque l'on désire instancier la classe Personne, nous faisons un appel a cette méthode par le mot clé new :

```
var _personne :Personne = new Personne("Dupont","Albert",1949) ;
var _personne_2 :Personne = new Personne("Dupont","Robert",1949) ;

_personne.egalite(_personne_2) ; -> retournera faux.
var agePersonne_2 :Number = _personne_2.calculAge() ;
```

Le principale avantage de la programmation orientée objet est de pouvoir hériter d'une classe déjà existante, si cela reste un concept bien intégré à ActionScript, il fait partie des notions de programmation orientée objet et ne s'applique pas qu'à ActionScript, nous ne le verrons donc pas.