9

Etendre les classes natives

INTÉRÊTS **1**
LE VIEIL AMI PROTOTYPE 2
ETENDRE LES CLASSES NON GRAPHIQUES **5**
ETENDRE LES CLASSES GRAPHIQUES **11**
ACCÉDER À L'OBJET STAGE DE MANIÈRE SÉCURISÉE 19
AJOUTER DES FONCTIONNALITÉS 23
RÉUTILISER LE CODE 47
CLASSE DYNAMIQUE 49
UN VRAI CONSTRUCTEUR 51
CRÉER DES BOUTONS DYNAMIQUES 52

Intérêts

Nous avons découvert au cours du chapitre précédent les principaux concepts clé de la programmation orientée objet. Cependant, une des principales difficultés réside souvent dans la mise en application de ces notions dans un projet concret ActionScript.

La grande puissance de Flash réside dans sa capacité à lier graphisme et programmation. Nous allons profiter de cette force pour appliquer ce que nous avons appris au cours du chapitre précédent à travers différents cas pratiques.

La notion d'héritage ne se limite pas aux classes personnalisées et peut être appliquée à n'importe quelle classe de l'API du lecteur Flash. Il est par exemple possible d'étendre la classe native `MovieClip`, en définissant de nouvelles méthodes afin d'obtenir un `MovieClip` amélioré.

Le but de ce chapitre est d'apprendre à étendre les classes natives telles `MovieClip`, `Sprite`, `BitmapData` mais aussi des classes non graphiques comme `Array` afin d'augmenter les fonctionnalités offertes par celle-ci.

Attention, nous verrons que certaines classes natives sont considérées comme scellées et ne peuvent être étendues.

Le vieil ami prototype

En ActionScript 1, les développeurs avaient pour habitude d'utiliser le `prototype` d'une classe afin d'augmenter ses capacités. En ActionScript 3, cette pratique fonctionne toujours mais n'est pas *officiellement* recommandée.

Dans le code suivant nous définissons une méthode `hello` sur le `prototype` de la classe `MovieClip` :

```
// ajout d'une méthode hello
MovieClip.prototype.hello = function ()

{

    // affiche : [object MovieClip]
    trace( this );

}

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

// le clip possède automatiquement la méthode ajoutée au prototype
monClip.hello();
```

Automatiquement, le clip crée possède la méthode `hello`. Bien qu'efficace, cette technique pollue les autres instances de la même classe, car la méthode `hello` est alors disponible sur tous les `MovieClip` de l'animation.

Dans le code suivant nous appelons la méthode `hello` sur le scénario principal :

```
// ajout d'une méthode hello
MovieClip.prototype.hello = function ()

{

    // affiche : [object MainTimeline]
    trace( this );

}

// le scénario possède automatiquement la méthode ajoutée au prototype
this.hello();
```

Dans un concept d'héritage, l'idée est d'obtenir un nouveau type d'objet, doté de nouvelles fonctionnalités. En utilisant cette technique nous ne créons aucune nouvelle variété d'objet, nous ajoutons simplement des fonctionnalités à une classe existante.

Imaginons que nous devions créer une balle ayant la capacité de rebondir de manière élastique. Nous pourrions définir sur le `prototype` de la classe `MovieClip` toutes les méthodes de collision nécessaires à notre balle, mais serait-ce réellement optimisé ?

En utilisant cette approche, tous les `MovieClip` de notre application seraient dotés de capacités de rebond. Pourtant, seule la balle a véritablement besoin de telles capacités. Il serait plus intéressant d'étendre la classe `MovieClip` et de lier notre balle à la sous-classe.

Le prototypage possède en revanche un intérêt majeur lié à sa simplicité et son efficacité. Prenons le cas de la classe `DisplayObjectContainer`.

Comme nous l'avons vu lors du chapitre 4 intitulé *Liste d'affichage*, la classe `DisplayObjectContainer` ne définit pas de méthode permettant de supprimer la totalité des objets enfants.

A l'aide d'une méthode ajoutée au prototype de la classe `DisplayObjectContainer` nous pouvons ajouter très facilement cette fonctionnalité :

```
DisplayObjectContainer.prototype.supprimeEnfants = function ( )
{
    var nbEnfants:int = this.numChildren;
    while ( this.numChildren > 0 ) this.removeChildAt ( 0 );
    return nbEnfants;
}
```

Ainsi, toutes les instances de la classe `DisplayObjectContainer` diposent désormais d'une méthode `supprimeEnfants`. En plus de permettre la suppression de tous les enfants, celle-ci retourne le nombre d'enfants supprimés.

Si nous disposons plusieurs objets graphiques sur le scénario principal, nous pouvons les supprimer en appelant la méthode `supprimeEnfants` :

```
DisplayObjectContainer.prototype.supprimeEnfants = function ( )
{
```

```
    var nbEnfants:int = this.numChildren;

    while ( this.numChildren > 0 ) this.removeChildAt ( 0 );

    return nbEnfants;

}

// supprime tous les objets enfants du scénario principal
this.supprimeEnfant();
```

De la même manière nous pouvons supprimer tous les objets enfants d'une instance de la classe `Sprite` :

```
// création d'un conteneur de type Sprite
var monSprite:Sprite = new Sprite ();

// création d'un objet Shape enfant
var maForme:Shape = new Shape();
maForme.graphics.lineStyle ( 1, 0x990000, 1 );
maForme.graphics.beginFill ( 0x990000, .2 );
maForme.graphics.drawCircle ( 50, 50, 50 );

monSprite.addChild( maForme );

addChild ( monSprite );

// suppression des objets enfants
var nbEnfantsSupprimes:int = monSprite.supprimeEnfants();
```

En testant le code précédent, une erreur à la compilation est générée :

```
1061: Appel à la méthode supprimeEnfants peut-être non définie, via la
référence de type static flash.display:Sprite.
```

Le compilateur empêche la compilation car aucune méthode du nom de `supprimeEnfants` n'est trouvée. Afin de faire taire le compilateur nous pouvons exceptionellement transtyper vers la classe dynamique `Object` non soumise à la vérification de type à la compilation :

```
// création d'un conteneur de type Sprite
var monSprite:Sprite = new Sprite ();

// création d'un objet Shape enfant
var maForme:Shape = new Shape();
maForme.graphics.lineStyle ( 1, 0x990000, 1 );
maForme.graphics.beginFill ( 0x990000, .2 );
maForme.graphics.drawCircle ( 50, 50, 50 );

monSprite.addChild( maForme );

addChild ( monSprite );

// suppression des objets enfants
var nbEnfantsSupprimes:int = Object(monSprite).supprimeEnfants();

// affiche : 1
trace( nbEnfantsSupprimes );
```

Nous reviendrons sur l'intérêt du prototypage au cours du chapitre 16 intitulé *Le texte*.

A retenir

- L'utilisation du prototype est toujours possible en ActionScript 3.
- Son utilisation est *officiellement* déconseillée, mais offre une souplesse intéressante dans certains cas précis.
- Nous préférerons dans la majorité des cas l'utilisation de sous-classes afin d'étendre les capacités.

Etendre les classes non graphiques

D'autres classes natives peuvent aussi être étendues afin d'augmenter leurs capacités, c'est le cas de la classe `Array`, dont les différentes méthodes ne sont quelquefois pas suffisantes.

Ne vous est-il jamais arrivé de vouloir rapidement mélanger les données d'un tableau ?

En étendant la classe `Array` nous allons ajouter un ensemble de méthodes pratiques, qui seront disponibles pour toute instance de sous-classe. A côté d'un nouveau document Flash CS3 nous définissons une classe `MonTableau` au sein du paquetage `org.bytearray.outils`.

Voici le code de la sous-classe `MonTableau` :

```
package org.bytearray.outils

{

    dynamic public class MonTableau extends Array

    {

         public function MonTableau ( ...rest )

         {


         }

    }

}
```

La sous-classe `MonTableau` est une classe dynamique car l'accès aux données par l'écriture crochet requiert l'utilisation d'une classe dynamique.

Etendre la classe `Array` nécessite une petite astuce qui n'a pas été corrigée avec ActionScript 3. Au sein du constructeur de la sous-classe nous devons ajouter la ligne suivante :

```
package org.bytearray.outils

{

    dynamic public class MonTableau extends Array

    {

          public function MonTableau ( ...rest )

          {

               splice.apply(this, [0, 0].concat(rest));

          }

    }

}
```

Cette astuce permet d'initialiser correctement le tableau lorsque des paramètres sont passés au constructeur.

A ce stade, nous bénéficions d'une sous-classe opérationnelle, qui possède toutes les capacités d'un tableau standard :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau (58, 48, 10);

// affiche : 48
trace( premierTableau[1] );

// ajout d'une valeur
premierTableau.push ( 25 );

// affiche : 4
trace( premierTableau.length );

// affiche : 25
trace( premierTableau.pop() ) ;

// affiche : 3
trace( premierTableau.length );
```

Pour l'instant, l'utilisation de la sous-classe `MonTableau` n'est pas vraiment justifiée, celle-ci ne possède aucune fonctionnalité en plus de la classe `Array`.

Afin de copier un tableau, nous pourrions être tentés d'écrire le code suivant :

```
// création d'un premier tableau
```

```
var monTableau:Array = new Array (58, 48, 10);

// création d'une copie ?
var maCopie:Array = monTableau;
```

Cette erreur fait référence à la notion de variables composites et primitives que nous avons traité lors du chapitre 2 intitulé *Langage et API du lecteur Flash*.

> Souvenez-vous, lors de la copie de données composites, nous copions les variables par référence et non par valeur.

Afin de créer une vraie copie de tableau nous définissons une nouvelle méthode `copie` au sein de la classe `MonTableau` :

```
package org.bytearray.outils

{

    dynamic public class MonTableau extends Array

    {

         public function MonTableau ( ...rest )

         {

              splice.apply(this, [0, 0].concat(rest));

         }

         public function copie ( ):MonTableau

         {

              var maCopie:MonTableau = new MonTableau();
              var lng:int = length;

              for ( var i:int = 0; i< lng; i++ ) maCopie[i] = this[i];

              return maCopie;

         }

    }

}
```

En testant le code suivant, nous voyons qu'une réelle copie du tableau est retournée par la méthode `copie` :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var tableau:MonTableau = new MonTableau (58, 48, 10);

// création d'une copie
```

```
var copieTableau:MonTableau = tableau.copie();

// modification d'une valeur
copieTableau[0] = 100;

// affiche : 12,58,85 100,58,85
trace ( tableau, copieTableau );
```

Nous aurions pu utiliser la méthode `Array.slice` mais celle-ci nous aurait renvoyé un tableau de type `Array` et non pas une nouvelle instance de `MonTableau`. En modifiant une valeur du tableau retourné nous voyons que les deux tableaux sont bien distincts.

Bien entendu, cette méthode ne fonctionne que pour la copie de tableaux contenant des données de types primitifs. Si nous souhaitons copier des tableaux contenant des données de types composites nous utiliserons la méthode `writeObject` de la classe `flash.utils.ByteArray`. Nous reviendrons sur cette fonctionnalité au cours du chapitre 20 intitulé *ByteArray*.

Nous allons maintenant ajouter une nouvelle méthode `melange` permettant de mélanger les données du tableau :

```
 public function melange ( ):void

{

    var i:int = length;
    var aleatoire:int;
    var actuel:*;

    while ( i-- )

    {

         aleatoire = Math.floor ( Math.random()*length );
         actuel = this[i];
         this[i] = this[aleatoire];
         this[aleatoire] = actuel;

    }

}
```

Dans le code suivant nous mélangeons différentes valeurs, nous pourrions utiliser cette méthode dans un jeu. Nous pourrions imaginer une série de questions stockées dans un tableau, à chaque lancement le tableau est mélangé afin que les joueurs n'aient pas les questions dans le même ordre :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var tableau:MonTableau = new MonTableau(58, 48, 10);

// mélange les valeurs
```

```
tableau.melange();

// affiche : 85,12,58
trace( tableau );
```

La méthode `egal` nous permet de tester si deux tableaux contiennent les mêmes valeurs :

```
public function egal ( pTableau:Array ):Boolean
{

    if ( this == pTableau ) return true;
    var i:int = this.length;
    if ( i != pTableau.length ) return false;
    while( i-- )
    {
         if ( this[i] != pTableau[i] ) return false;
    }
    return true;

}
```

Dans le code suivant nous comparons deux tableaux :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(12, 58, 85);

// création d'un autre tableau de nombres
var secondTableau:MonTableau = new MonTableau(12, 58, 85);

// affiche : true
trace( premierTableau.egal ( secondTableau ) );
```

Lorsque le tableau passé contient les mêmes valeurs, la méthode `egal` renvoie `true`. A l'inverse si nous modifions les données du premier tableau, la méthode `egal` renvoie `false` :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(100, 58, 85);

// création d'un autre tableau de nombres
var secondTableau:MonTableau = new MonTableau(12, 58, 85);

// affiche : false
trace( premierTableau.egal ( secondTableau ) );
```

Dans cet exemple nous ne prenons pas en charge les tableaux à plusieurs dimensions, différentes approches pourraient être utilisées, à vous de jouer !

Il serait pratique d'avoir une méthode qui se chargerait de calculer la moyenne des valeurs contenues dans le tableau. Avant de démarrer le

calcul nous devons nous assurer que le tableau ne contienne que des nombres.

Pour cela nous utilisons la nouvelle méthode `every` de la classe `Array` :

```
public function moyenne ( ):Number

{

    if ( ! every ( filtre ) ) throw new TypeError ("Le tableau actuel ne
contient pas que des nombres");

    var i:int = length;
    var somme:Number = 0;

    while ( i-- ) somme += this[i];

    return somme/length;

}

private function filtre ( pElement:*, pIndex:int, pTableau:Array ):Boolean

{

    return pElement is Number;

}
```

Lorsque la méthode `moyenne` est exécutée, celle-ci vérifie dans un premier temps si la totalité des données du tableau sont bien des nombres. Pour cela la méthode `filtre` est exécutée sur chaque élément du tableau, et renvoie `true` tant que l'élément parcouru est un nombre. Si ce n'est pas le cas, celle-ci renvoie `false` et nous levons une erreur de type `TypeError`. Si aucune erreur n'est levée, nous pouvons entamer le calcul de la moyenne.

Dans le code suivant, le calcul est possible :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(100, 58, 85);

// affiche : 81
trace( premierTableau.moyenne() );
```

A l'inverse, si une valeur du tableau n'est pas un nombre :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(this, 58, false);

trace( premierTableau.moyenne() );
```

Une erreur à l'exécution est levée :

```
TypeError: Le tableau actuel ne contient pas que des nombres
```

Afin de gérer l'erreur, nous pouvons placer l'appel de la méthode `moyenne` au sein d'un bloc `try catch` :

```
// import de la classe MonTableau
import org.bytearray.outils.MonTableau;

// création d'un tableau de nombres
var premierTableau:MonTableau = new MonTableau(this, 58, false);

try

{

    trace( premierTableau.moyenne() );

} catch ( pError:Error )

{

    trace("une erreur de calcul est survenue !");

}
```

Nous pourrions ajouter autant de méthodes que nous souhaitons au sein de la classe `MonTableau`, libre à vous d'ajouter les fonctionnalités dont vous avez besoin. Cela nous permet de substituer la classe `MonTableau` à la classe `Array` afin de toujours avoir à disposition ces fonctionnalités.

A retenir

- Toutes les classes natives ne sont pas sous-classables.
- La composition peut être utilisée afin d'augmenter les capacités d'une classe qui n'est pas sous-classable.
- Etendre une classe native est un moyen élégant d'étendre les capacités d'ActionScript 3 ou du lecteur.

Etendre les classes graphiques

L'extension de classes natives prend tout son sens dans le cas de sous-classes graphiques. Nous avons développé une application de dessin au cours du chapitre 7 intitulé *Intéractivité*, nous allons à présent l'enrichir en ajoutant un objet graphique tel un stylo.

Le graphisme a été réalisé sous 3D Studio Max puis ensuite exporté en image pour être utilisé dans Flash. Dans un nouveau document Flash CS3 nous importons le graphique puis nous le transformons en symbole clip.

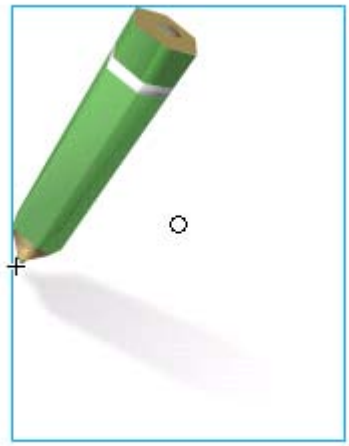

Figure 9-1. Graphique du stylo converti en symbole clip.

Attention à bien modifier le point d'enregistrement, de manière à ce que la mine du stylo soit en coordonnée 0,0.

Ce clip va nous servir de graphisme afin de représenter le stylo de l'application. Nous devons pour cela le rendre disponible par programmation. Au sein du panneau *Propriétés de liaison* nous cochons la case *Exporter pour ActionScript* et renseignons `Stylo` comme nom de classe, nous laissons `flash.display.MovieClip` comme classe de base.

La classe `Stylo` est donc une sous-classe de `MovieClip` :

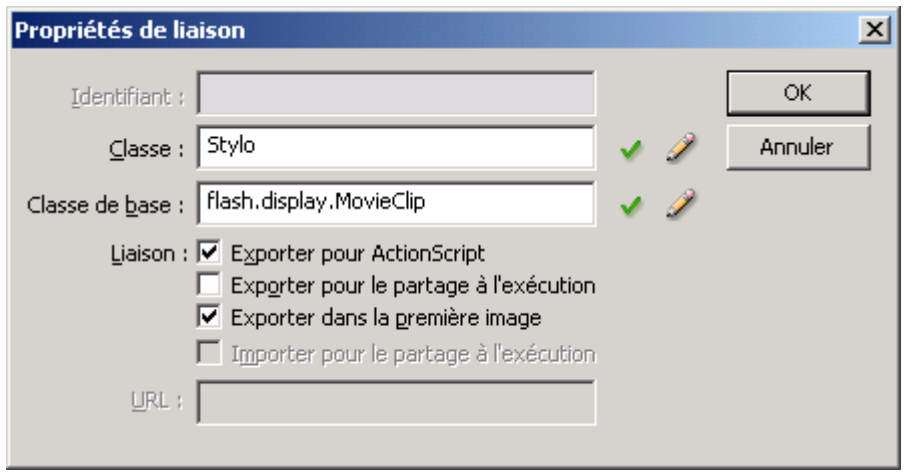

Figure 9-2. Panneau propriétés de liaison.

Lorsque nous cliquons sur le bouton OK, Flash tente de trouver une classe du même nom qui aurait pu être définie. Si il n'en trouve aucune, Flash affiche le message illustré en figure 9-3 :

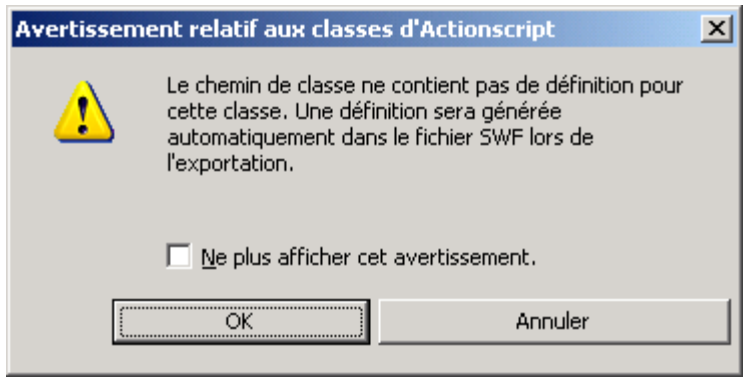

Figure 9-3. Génération automatique de classe.

Comme nous l'avons vu au cours du chapitre 5 intitulé *Les symboles*, Flash génère automatiquement une classe interne utilisée pour instancier le symbole.

Dans notre cas, une classe `Stylo` est générée par Flash, afin de pouvoir instancier notre stylo puis l'afficher de la manière suivante :

```
// instanciation du stylo
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
addChild ( monStylo );
```

Flash génère automatiquement une classe `Stylo` qui hérite de la classe `MovieClip`, rappelez-vous que cette classe est inaccessible.

Si nous avions accès à celle-ci nous pourrions lire le code suivant :

```
package

{

    import flash.display.MovieClip;

    public class Stylo extends MovieClip

    {

         public function Stylo ()

         {


         }

    }
```

```
}
```

La classe `Stylo` possède donc toutes les capacités d'un `MovieClip` :

```
// instanciation du stylo
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
addChild ( monStylo );

// la classe stylo possède toutes les capacités d'un clip
monStylo.stop();
monStylo.play();
monStylo.gotoAndStop (2);
monStylo.prevFrame ();
```

Cette génération automatique de classe s'avère très pratique lorsque nous ne souhaitons pas définir de classe manuellement pour chaque objet, Flash s'en charge et cela nous permet de gagner un temps précieux. En revanche Flash nous laisse aussi la possibilité de définir manuellement les sous classe graphiques, et d'y ajouter tout ce que nous souhaitons.

A coté de notre document Flash CS3 nous créons une classe `Stylo` héritant de `MovieClip` :

```
 package

{

    import flash.display.MovieClip;

    public class Stylo extends MovieClip

    {

          public function Stylo ()

          {

               trace( this );

          }

    }

}
```

Nous sauvons la classe sous le nom `Stylo.as`, attention, le nom de la classe doit être le même que le fichier `.as`.

L'organisation de nos fichiers de travail doit être la suivante :

Figure 9-4. Organisation des fichiers.

Flash va désormais utiliser notre définition de classe et non celle générée automatiquement. Pour s'en assurer, nous utilisons le panneau *Propriétés de liaison*.

A droite du champ *Classe* sont situées deux icônes. En cliquant sur la première, nous pouvons valider la définition de la classe afin de vérifier que Flash utilise bien notre définition de classe.

La figure 9-5 illustre les deux icônes :

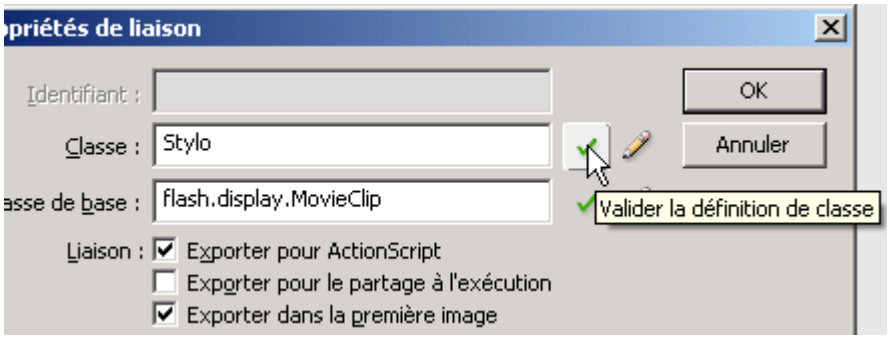

Figure 9-5. Organisation des fichiers.

Une fois la définition de classe validée, un message nous indiquant que la classe a bien été détectée s'affiche. Si nous cliquons sur l'icône d'édition représentant un stylo située à droite de l'icône de validation, la classe s'ouvre au sein de Flash afin d'être éditée :

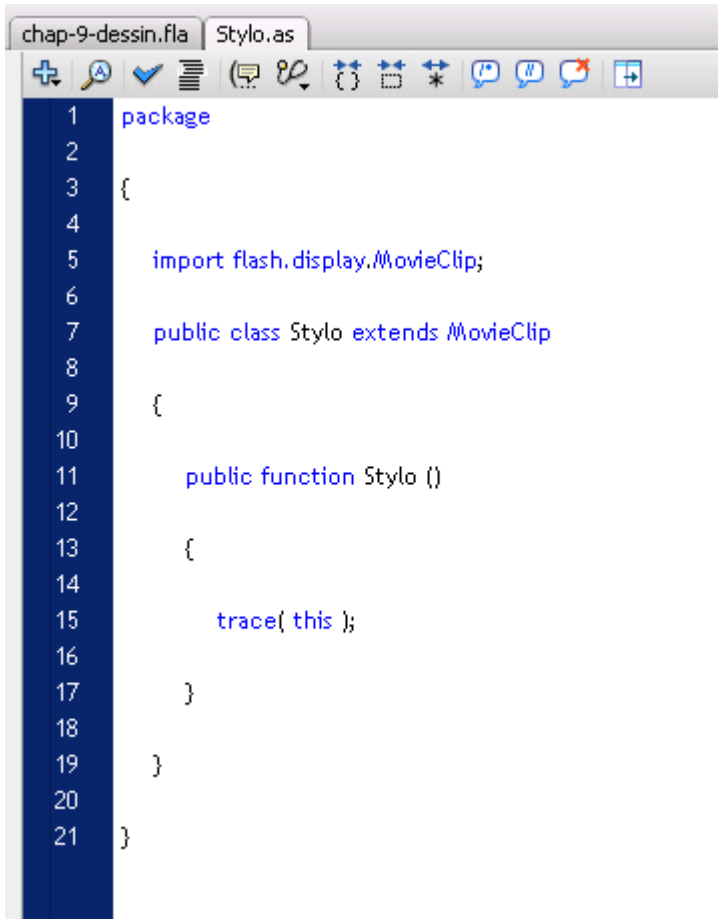

Figure 9-6. Edition de la classe au sein de Flash CS3.

Le symbole est correctement lié à notre classe, nous pouvons dès à présent l'instancier :

```
// affiche : [object Stylo]
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
addChild ( monStylo );

// positionnement en x et y
monStylo.x = 250;
monStylo.y = 200;
```

Lorsque le stylo est crée le constructeur est déclenché, l'instruction `trace` que nous avions placé affiche : `[object Stylo]`.

Notre symbole est affiché comme l'illustre la figure 9-7 :

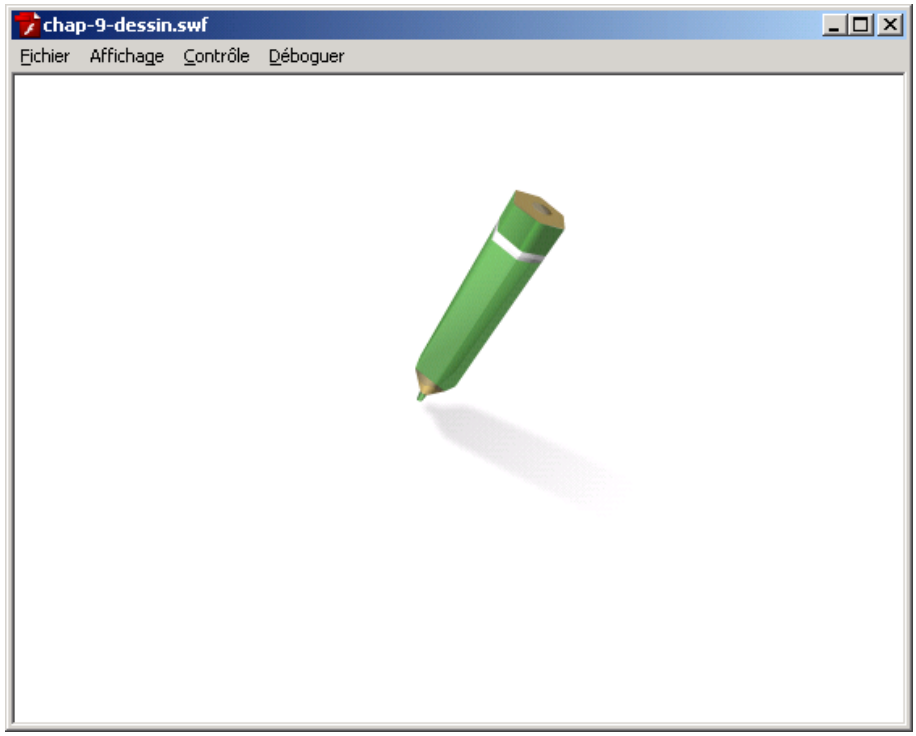

Figure 9-7. Symbole Stylo affiché.

Nous n'avons défini pour le moment aucune nouvelle fonctionnalité au sein de la classe `Stylo`, celle-ci étend simplement la classe `flash.display.MovieClip`.

Toutes les fonctionnalités que nous ajouterons au sein de la classe `Stylo` seront automatiquement disponibles au sein du symbole, celui-ci est désormais *lié à la classe*.

En ajoutant une méthode `test` au sein de la classe `Stylo` :

```
package
{
  import flash.display.MovieClip;
  public class Stylo extends MovieClip
  {
       public function Stylo ()
       {
            trace( this );
       }
       public function test ( ):void
```

```
        {

            trace("ma position dans l'axe des x est de : " + x );
            trace("ma position dans l'axe des y est de : " + y );

        }

    }

}
```

Celle-ci est automatiquement disponible auprès de l'instance :

```
// affiche : [object Stylo]
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
addChild ( monStylo );

// positionnement en x et y
monStylo.x = 250;
monStylo.y = 200;

/* affiche :
ma position dans l'axe des x est de : 250
ma position dans l'axe des y est de : 200
*/
monStylo.test();
```

La première chose que notre stylo doit savoir faire est de suivre la souris. Nous n'allons pas utiliser l'instruction `startDrag` car nous verrons que nous devons travailler sur le mouvement du stylo, l'instruction `startDrag` ne nous le permettrait pas. Pour cela nous allons utiliser l'événement `MouseEvent.MOUSE_MOVE`.

Notre classe est une sous-classe de `MovieClip` et possède donc tous les événements interactifs nécessaires dont nous avons besoin, au sein du constructeur nous écoutons l'événement `MouseEvent.MOUSE_MOVE` :

```
package

{

  import flash.display.MovieClip;
  import flash.events.MouseEvent;

  public class Stylo extends MovieClip

  {

        public function Stylo ()

        {

            trace( this );

            addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
```

```
            }

            private function bougeSouris ( pEvt:MouseEvent ):void

            {

                  trace( pEvt );

            }

      }

}
```

Dans le code précédent la méthode `bougeSouris` est à l'écoute de l'événement `MouseEvent.MOUSE_MOVE`.

Pourquoi préférons-nous l'évenement `MouseEvent.MOUSE_MOVE` à l'évenement `Event.ENTER_FRAME` ?

Ce dernier se déclenche en continu indépendamment du mouvement de la souris. Ainsi, même si le stylo est immobile l'événement `Event.ENTER_FRAME` est diffusé. A terme, cela pourrait ralentir les performances de notre application.

Souvenez-vous que l'événement `MouseEvent.MOUSE_MOVE` n'est plus global comme en ActionScript 1 et 2, et n'est diffusé que lors du survol du stylo. Si nous avons besoin d'écouter le mouvement de la souris sur toute la scène nous devons accéder à l'objet `Stage`.

A retenir

- Il est possible de lier un symbole existant à une sous-classe graphique définie manuellement.
- Le symbole hérite de toutes les fonctionnalités de la sous-classe.

Accéder à l'objet Stage de manière sécurisée

Comme nous l'avons vu lors du chapitre 7 intitulé *Interactivité*, seul l'objet `Stage` permet une écoute globale de la souris ou du clavier. Nous devons donc modifier notre code afin d'écouter l'événement `MouseEvent.MOUSE_MOVE` auprès de l'objet `Stage` :

```
package

{

  import flash.display.MovieClip;
  import flash.events.MouseEvent;

  public class Stylo extends MovieClip

  {
```

```
        public function Stylo ()
        {
            trace( this );
            stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
        }
        private function bougeSouris ( pEvt:MouseEvent ):void
        {
            trace( pEvt );
        }
    }
}
```

En testant le code précédent, nous obtenons l'erreur suivante à l'exécution :

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

Souvenez-vous, lors du chapitre 4 intitulé *La liste d'affichage* nous avons vu que la propriété `stage` propre à tout objet de type `flash.display.DisplayObject` renvoyait `null` tant que l'objet graphique n'était pas ajouté à la liste d'affichage.

Ainsi, lorsque nous créons le symbole :

```
var monStylo:Stylo = new Stylo();
```

Le constructeur est déclenché et nous tentons alors d'accéder à l'objet `Stage`. A ce moment là, notre symbole n'est pas encore présent au sein de la liste d'affichage, la propriété `stage` renvoie donc `null` et l'appel à la méthode `addEventListener` échoue.

Comment allons-nous procéder ?

Lors du chapitre 4 intitulé *La liste d'affichage* nous avons découvert deux événements importants liés à l'activation et à la désactivation des objets graphiques.

Voici un rappel de ces deux événements fondamentaux :

- `Event.ADDED_TO_STAGE` : cet événement est diffusé lorsque l'objet graphique est placé au sein d'un `DisplayObjectContainer` présent au sein de la liste d'affichage.
- `Event.REMOVED _FROM_STAGE` : cet événement est diffusé lorsque l'objet graphique est supprimé de la liste d'affichage.

Nous devons attendre que le symbole soit ajouté à la liste d'affichage pour pouvoir accéder à l'objet `Stage`. Le symbole va se souscrire lui-même auprès de l'événement `Event.ADDED_TO_STAGE` qu'il diffusera lorsqu'il sera ajouté à la liste d'affichage. Cela peut paraître étrange, mais dans ce cas l'objet s'écoute lui-même.

Nous modifions la classe `Stylo` afin d'intégrer ce mécanisme :

```
package

{

  import flash.display.MovieClip;
  import flash.events.Event;

  public class Stylo extends MovieClip

  {

        public function Stylo ()

        {

              trace( this );

              // le stylo écoute l'événement Event.ADDED_TO_STAGE, diffusé
lorsque celui-ci est ajouté à la liste d'affichage
              addEventListener ( Event.ADDED_TO_STAGE, activation );

        }

        private function activation ( pEvt:Event ):void

        {

              trace( pEvt );

        }

  }

}
```

Lors d'instanciation du symbole `Stylo`, le constructeur est déclenché :

```
// affiche : [object Stylo]
var monStylo:Stylo = new Stylo();
```

Lorsque l'instance est ajoutée à la liste d'affichage, l'événement `Event.ADDED_TO_STAGE` est diffusé :

```
// affiche : [object Stylo]
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
// affiche : [Event type="addedToStage" bubbles=false cancelable=false
eventPhase=2]
addChild ( monStylo );
```

Nous définissons la méthode écouteur `activation` comme privée car nous n'y accèderons pas depuis l'extérieur. Gardez bien en tête de rendre privées toutes les méthodes et propriétés qui ne seront pas utilisées depuis l'extérieur de la classe.

L'événement `Event.ADDED_TO_STAGE` est diffusé, nous remarquons au passage que nous écoutons la phase cible et qu'il s'agit d'un événement qui ne participe pas à la phase de remontée.

Lorsque la méthode `activation` est déclenchée nous pouvons cibler en toute sécurité la propriété `stage` et ainsi écouter l'événement `MouseEvent.MOUSE_MOVE` :

```
package

{

  import flash.display.MovieClip;
  import flash.events.MouseEvent;
  import flash.events.Event;

  public class Stylo extends MovieClip

  {

        public function Stylo ()

        {

              // le stylo écoute l'événement Event.ADDED_TO_STAGE
              // diffusé lorsque celui-ci est ajouté à la liste d'affichage
              addEventListener ( Event.ADDED_TO_STAGE, activation );

        }

        private function activation ( pEvt:Event ):void

        {

              // écoute de l'événement MouseEvent.MOUSE_MOVE
              stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

        }

        private function bougeSouris ( pEvt:MouseEvent ):void

        {

              trace( pEvt );

        }

  }

}
```

Si nous testons le code précédent, nous remarquons que la méthode `bougeSouris` est bien déclenchée lorsque la souris est déplacée sur la totalité de la scène.

Ajouter des fonctionnalités

Nous allons à présent nous intéresser au mouvement du stylo en récupérant les coordonnées de la souris afin de le positionner.

Nous définissons deux propriétés `positionX` et `positionY` :

```
// position en cours de la souris
private var positionX:Number;
private var positionY:Number;
```

Celles-ci permettent de stocker la position de la souris :

```
private function bougeSouris ( pEvt:MouseEvent ):void

{

  // récupération des coordonnées de la souris en x et y
  positionX = pEvt.stageX;
  positionY = pEvt.stageY;

  // affectation de la position
  x = positionX;
  y = positionY;

}
```

Le stylo suit désormais la souris comme l'illustre la figure 9-8 :

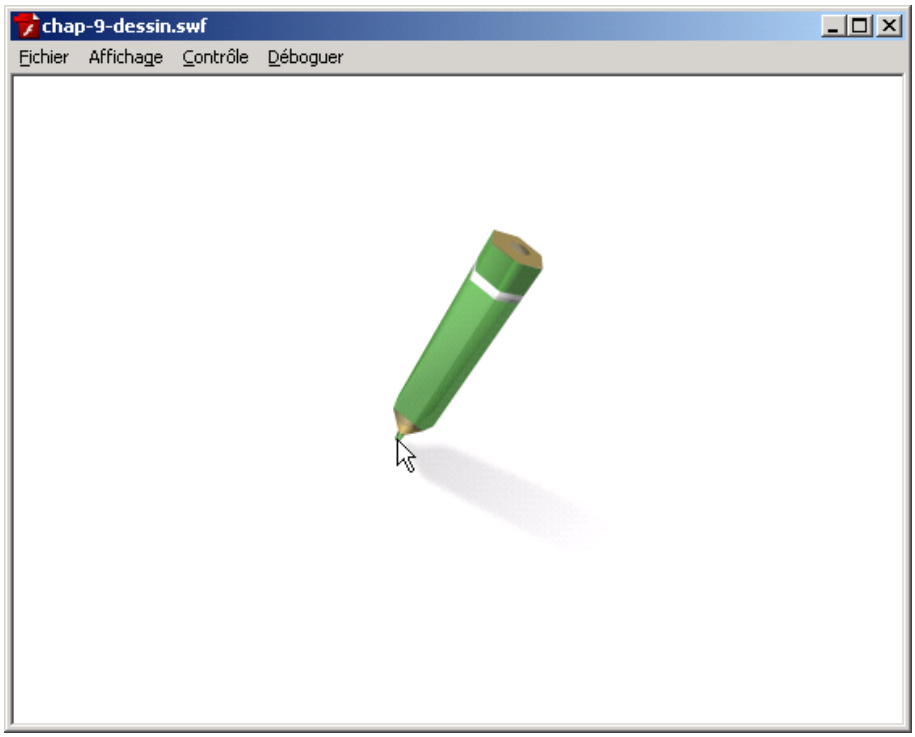

Figure 9-8. Stylo attaché au curseur.

En testant l'application nous remarquons que le mouvement n'est pas totalement fluide, nous allons utiliser la méthode

`updateAfterEvent` que nous avons étudié au cours du chapitre 7 intitulé *Interactivité*.

Souvenez-vous, cette méthode de la classe `MouseEvent` nous permet de forcer le rafraîchissement du lecteur :

```
private function bougeSouris ( pEvt:MouseEvent ):void

{

  // récupération des coordonnées de la souris en x et y
  positionX = pEvt.stageX;
  positionY = pEvt.stageY;

  // affectation de la position
  x = positionX;
  y = positionY;

  // force le rafraîchissement
  pEvt.updateAfterEvent();

}
```

Le mouvement est maintenant fluide mais le curseur de la souris demeure affiché, nous le masquons à l'aide de la méthode statique `hide` de la classe `Mouse` lors de l'activation de l'objet graphique.

> Veillez à ne pas placer cet appel en continu au sein de la méthode `bougeSouris` cela serait redondant.

Nous importons la classe `Mouse` :

```
import flash.ui.Mouse;
```

Puis nous modifions la méthode `activation` :

```
private function activation ( pEvt:Event ):void

{

      // cache le curseur
      Mouse.hide();

      // écoute de l'événement MouseEvent.MOUSE_MOVE
      stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

}
```

A ce stade, nous avons étendu les capacités de la classe `MovieClip`, les symboles liés à la classe `Stylo` peuvent désormais suivrent la souris. Nous pouvons à tout moment lier notre classe `Stylo` à n'importe quel autre symbole celui-ci bénéficiera aussitôt des toutes les fonctionnalités définies par celle-ci.

Nous avons pourtant oublié un élément essentiel, voyez-vous de quoi il s'agit ?

Nous devons gérer la désactivation de l'objet graphique afin de libérer les ressources. Si nous supprimons le stylo de la liste d'affichage, l'événement `MouseEvent.MOUSE_MOVE` est toujours diffusé, le curseur souris demeure masqué :

```
var monStylo:Stylo = new Stylo();

// ajout à la liste d'affichage
// affiche : [Event type="addedToStage" bubbles=false cancelable=false
eventPhase=2]
addChild ( monStylo );

// suppression du stylo, celui ci n'intègre aucune logique de désactivation
removeChild ( monStylo );
```

Pour gérer proprement cela nous écoutons l'événement `Event.REMOVED_FROM_STAGE` :

```
public function Stylo ()

{

  // le stylo écoute l'événement Event.ADDED_TO_STAGE
  // diffusé lorsque celui-ci est ajouté à la liste d'affichage
  addEventListener ( Event.ADDED_TO_STAGE, activation );

  // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
  // diffusé lorsque celui-ci est supprimé de la liste d'affichage
  addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

}
```

Puis nous ajoutons une méthode écouteur `desactivation` afin d'intégrer la logique de désactivation nécessaire :

```
private function desactivation ( pEvt:Event ):void

{

  // affiche le curseur
  Mouse.show();

  // arrête l'écoute de l'événement MouseEvent.MOUSE_MOVE
  stage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

}
```

Il serait intéressant d'ajouter un effet de rotation au stylo lorsque celui-ci arrive en haut de la scène afin de simuler une rotation du poignet.

La figure 9-9 illustre l'idée :

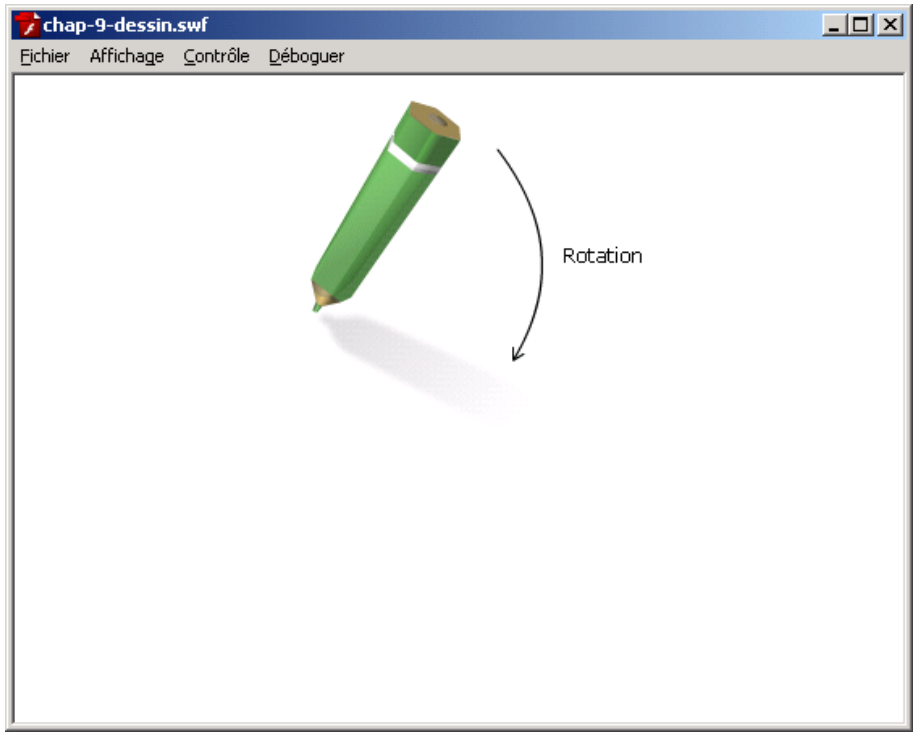

Figure 9-9. Rotation du stylo.

En intégrant une simple condition nous allons pouvoir donner plus de réalisme au mouvement du stylo. Nous allons dans un premier temps définir la position dans l'axe des y à partir de laquelle le stylo commence à s'incliner, pour cela nous définissons une propriété constante car celle-ci ne changera pas à l'exécution :

```
// limite pour l'axe des y
private static const LIMIT_Y:int = 140;
```

Puis nous intégrons la condition au sein de la méthode `bougeSouris` :

```
private function bougeSouris ( pEvt:MouseEvent ):void

{

  // récupération des coordonnées de la souris en x et y
  positionX = pEvt.stageX;
  positionY = pEvt.stageY;

  // affectation de la position
  x = positionX;
  y = positionY;

  // si le stylo passe dans la zone supérieure alors nous inclinons le stylo
  if ( positionY < Stylo.LIMIT_Y )

  {

       rotation = ( Stylo.LIMIT_Y - positionY );
```

```
        }

        // force le rafraîchissement
        pEvt.updateAfterEvent();

    }
```

A ce stade, voici le code complet de notre classe `Stylo` :

```
package

{

    import flash.display.MovieClip;
    import flash.events.MouseEvent;
    import flash.events.Event;
    import flash.ui.Mouse;

    public class Stylo extends MovieClip

    {

        // limite pour l'axe des y
        private static const LIMIT_Y:int = 140;

        // position en cours de la souris
        private var positionX:Number;
        private var positionY:Number;

        public function Stylo ()

        {

            // le stylo écoute l'événement Event.ADDED_TO_STAGE
            // diffusé lorsque celui-ci est ajouté à la liste d'affichage
            addEventListener ( Event.ADDED_TO_STAGE, activation );

            // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
            // diffusé lorsque celui-ci est supprimé de la liste d'affichage
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

        }

        private function activation ( pEvt:Event ):void

        {

            // cache le curseur
            Mouse.hide();

            // écoute de l'événement MouseEvent.MOUSE_MOVE
            stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

        }

        private function desactivation ( pEvt:Event ):void

        {

            // affiche le curseur
            Mouse.show();
```

```
            // arrête l'écoute de l'événement MouseEvent.MOUSE_MOVE
            stage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );

      }

      private function bougeSouris ( pEvt:MouseEvent ):void

      {

            // récupération des coordonnées de la souris en x et y
            positionX = pEvt.stageX;
            positionY = pEvt.stageY;

            // affectation de la position
            x = positionX;
            y = positionY;

            // si le stylo passe dans la zone supérieure alors nous inclinons
le stylo
            if ( positionY < Stylo.LIMIT_Y )

            {

                  rotation = ( Stylo.LIMIT_Y - positionY );

            }

            // force le rafraîchissement
            pEvt.updateAfterEvent();

      }

  }

}
```

Si nous testons l'application, lorsque nous arrivons en zone supérieure le stylo s'incline comme l'illustre la figure 9-10 :

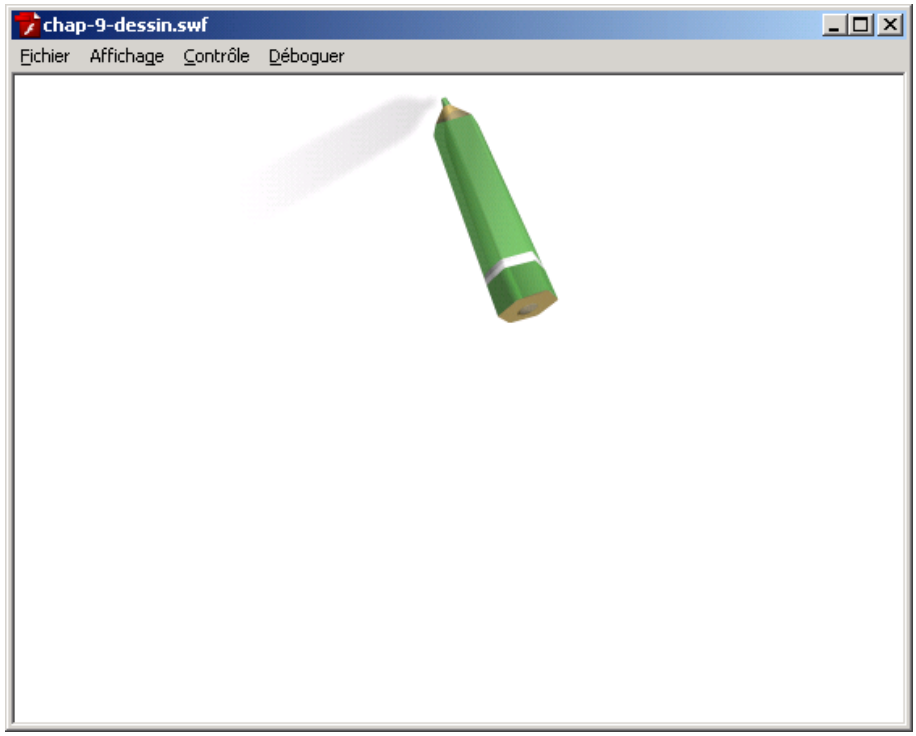

Figure 9-10. Inclinaison du stylo.

Afin de rendre plus naturel le mouvement du stylo, nous ajoutons une formule d'inertie :

```
private function bougeSouris ( pEvt:MouseEvent ):void

{

  // récupération des coordonnées de la souris en x et y
  positionX = pEvt.stageX;
  positionY = pEvt.stageY;

  // affectation de la position
  x = positionX;
  y = positionY;

  // si le stylo passe dans la zone supérieure alors nous inclinons le stylo
  if ( positionY < Stylo.LIMIT_Y )

  {

       rotation -= ( rotation - ( Stylo.LIMIT_Y - positionY ) ) *.2;

  // sinon, le stylo reprend son inclinaison d'origine
  } else rotation -= ( rotation - 0 ) *.2;

  // force le rafraîchissement
  pEvt.updateAfterEvent();

}
```

Il est temps d'ajouter à présent la notion de dessin, cette partie a déjà été abordée lors du chapitre 7, nous allons donc réutiliser ce code dans notre application.

Si nous pensons en termes de séparation des tâches, il paraît logique qu'un stylo se charge de son mouvement et du dessin, et non pas à la création de la toile où dessiner. Nous allons donc définir une méthode `affecteToile` qui aura pour mission d'indiquer au stylo dans quel conteneur dessiner.

Nous définissons dans la classe une propriété permettant de référencer le conteneur :

```
// stocke une référence au conteneur de dessin
private var conteneur:DisplayObjectContainer;
```

Puis nous importons la classe `flash.display.DisplayObjectContainer` :

```
import flash.display.DisplayObjectContainer;
```

Et ajoutons la méthode `affecteToile` :

```
// méthode permettant de spécifier le conteneur du dessin
public function affecteToile ( pToile:DisplayObjectContainer ):void

{

     conteneur = pToile;

}
```

Tout type de conteneur peut être passé, à condition que celui-ci soit de type `DisplayObjectContainer` :

```
// création du conteneur de tracés vectoriels
var toile:Sprite = new Sprite();

// ajout du conteneur à la liste d'affichage
addChild ( toile );

// création du symbole
var monStylo:Stylo = new Stylo();

// affectation du conteneur de tracés
monStylo.affecteToile ( toile );

// ajout du symbole à la liste d'affichage
addChild ( monStylo );

// positionnement en x et y
monStylo.x = 250;
monStylo.y = 200;
```

Ainsi, différents types d'objets graphiques peuvent servir de toile. Souvenez-vous grâce à l'héritage un sous type peut être passé partout où un super-type est attendu.

En plus de réutiliser les fonctionnalités de dessin que nous avions développé durant le chapitre 7 intitulé *Interactivité* nous allons aussi réutiliser la notion d'historique combinés aux raccourcis clavier : CTRL+Z et CTRL+Y.

Nous définissons trois nouvelles propriétés permettant de stocker les formes crées et supprimées, puis le tracé en cours :

```
// tableaux référençant les formes tracées et supprimées
private var tableauTraces:Array = new Array();
private var tableauAncienTraces:Array = new Array();

// référence le tracé en cours
private var monNouveauTrace:Shape;
```

Attention, la classe `flash.display.Shape` doit être importée :

```
import flash.display.Shape;
```

Au sein de la méthode `activation`, nous devons tout d'abord écouter le clic souris, afin de savoir quand est-ce que l'utilisateur souhaite commencer à dessiner :

```
private function activation ( pEvt:Event ):void

{

  // cache le curseur
  Mouse.hide();

  // écoute des différents événements
  stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
  stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );

}
```

Nous ajoutons la méthode écouteur `clicSouris`, qui se charge de créer les objets et d'initialiser le style du tracé :

```
  private function clicSouris ( pEvt:MouseEvent ):void

  {

        if ( conteneur == null ) throw new Error ( "Veuillez appeler au
préalable la méthode affecteToile()" );

        // récupération des coordonnées de la souris en x et y
        positionX = pEvt.stageX;
        positionY = pEvt.stageY;

        // un nouvel objet Shape est crée pour chaque tracé
        monNouveauTrace = new Shape();

        // nous ajoutons le conteneur de tracé au conteneur principal
        conteneur.addChild ( monNouveauTrace );

        // puis nous référençons le tracé au sein du tableau
        // référençant les tracés affichés
        tableauTraces.push ( monNouveauTrace );
```

```
        // nous définissons un style de tracé
        monNouveauTrace.graphics.lineStyle ( 1, 0x990000, 1 );

        // la mine est déplacée à cette position
        // pour commencer à dessiner à partir de cette position
        monNouveauTrace.graphics.moveTo ( positionX, positionY );

        // si un nouveau tracé intervient alors que nous sommes
        // repartis en arrière nous repartons de cet état
        if ( tableauAncienTraces.length ) tableauAncienTraces = new Array;

        // écoute du mouvement de la souris
        stage.addEventListener ( MouseEvent.MOUSE_MOVE, dessine );

    }
```

Puis nous définissons la méthode `dessine` afin de gérer le tracé :

```
    private function dessine ( pEvt:MouseEvent ):void
    {

        if ( monNouveauTrace != null )

        {

            // la mine est déplaçée à cette position
            // pour commencer à dessiner à partir de cette position
            monNouveauTrace.graphics.lineTo ( positionX, positionY );

        }

    }
```

Afin d'arrêter de dessiner, nous écoutons le relâchement de la souris grâce à l'événement `MouseEvent.MOUSE_UP` :

```
    private function activation ( pEvt:Event ):void

    {

        // cache le curseur
        Mouse.hide();

        // écoute des différents événements
        stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
        stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
        stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );

    }
```

La méthode écouteur `relacheSouris` est déclenchée lors du relâchement de la souris et interrompt l'écoute de l'événement `MouseEvent.MOUSE_MOVE` :

```
    private function relacheSouris ( pEvt:MouseEvent ):void

    {

        stage.removeEventListener ( MouseEvent.MOUSE_MOVE, dessine );
```

```
}
```

Si nous testons notre application, tout fonctionne correctement, nous pouvons à présent dessiner, lorsque la souris est relâchée, le tracé est stoppé.

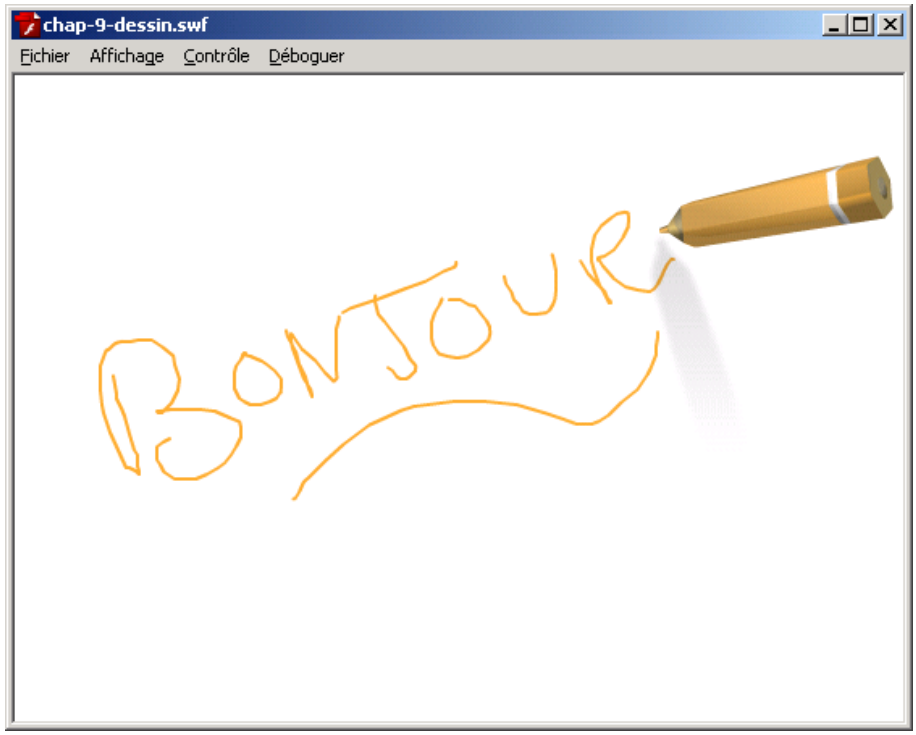

Figure 9-11. Application de dessin.

Notre application est bientôt terminée, nous devons ajouter la notion d'historique pour cela nous importons la classe `KeyboardEvent` :

```
import flash.events.KeyboardEvent;
```

Puis nous ajoutons l'écoute du clavier au sein de la méthode `activation` :

```
private function activation ( pEvt:Event ):void

{

     // cache le curseur
     Mouse.hide();

     // écoute de l'événement MouseEvent.MOUSE_MOVE
     stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
     stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
     stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );
     stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );

}
```

La méthode écouteur `ecouteClavier` se charge de gérer l'historique, nous importons la classe `Keyboard` :

```
import flash.ui.Keyboard;
```

Et définissons deux propriétés constantes stockant le code de chaque touche :

```
// code des touches Y et Z
private static const codeToucheY:int = 89;
private static const codeToucheZ:int = 90;
```

Puis nous ajoutons le code déjà développé durant le chapitre 7 pour la précédente application de dessin :

```
private function ecouteClavier ( pEvt:KeyboardEvent ):void

{

	// si la barre espace est enfoncée
	if ( pEvt.keyCode == Keyboard.SPACE )

	{

		// nombre d'objets Shape contenant des tracés
		var lng:int = tableauTraces.length;

		// suppression des traçés de la liste d'affichage
		while ( lng-- ) conteneur.removeChild ( tableauTraces[lng] );

		// les tableaux d'historiques sont reinitialisés
		// les références supprimées
		tableauTraces = new Array();
		tableauAncienTraces = new Array();
		monNouveauTrace = null;

	}

	if ( pEvt.ctrlKey )

	{

		// si retour en arrière (CTRL+Z)
		if( pEvt.keyCode == Stylo.codeToucheZ && tableauTraces.length )

		{

			// nous supprimons le dernier tracé
			var aSupprimer:Shape = tableauTraces.pop();

			// nous stockons chaque tracé supprimé
			// dans le tableau spécifique
			tableauAncienTraces.push ( aSupprimer );

			// nous supprimons le tracé de la liste d'affichage
			conteneur.removeChild( aSupprimer );

		// si retour en avant (CTRL+Y)
		} else if ( pEvt.keyCode == Stylo.codeToucheY &&
tableauAncienTraces.length )
```

```
                {

                    // nous récupérons le dernier tracé ajouté
                    var aAfficher:Shape = tableauAncienTraces.pop();

                    // nous le replaçons dans le tableau de
                    // tracés à l'affichage
                    tableauTraces.push ( aAfficher );

                    // puis nous l'affichons
                    conteneur.addChild ( aAfficher );

                }

            }

        }
```

Nous obtenons un symbole stylo intelligent doté de différentes fonctionnalités que nous avons ajoutées. Un des principaux avantages des sous-classes graphiques réside dans leur facilité de réutilisation.

Nous pouvons à tout moment lier un autre symbole à notre sous-classe graphique, ce dernier héritera automatiquement des fonctionnalités de la classe `Stylo` et deviendra opérationnel.

Avant cela, nous allons ajouter une autre fonctionnalité liée à la molette souris. Il serait élégant de pouvoir choisir la couleur du tracé à travers l'utilisation de la molette souris. Lorsque l'utilisateur s'en servira, la couleur du tracé et du stylo sera modifiée.

N'oublions pas qu'au sein de la sous-classe nous sommes sur le scénario du symbole. Nous modifions le symbole stylo en ajoutant plusieurs images clés :

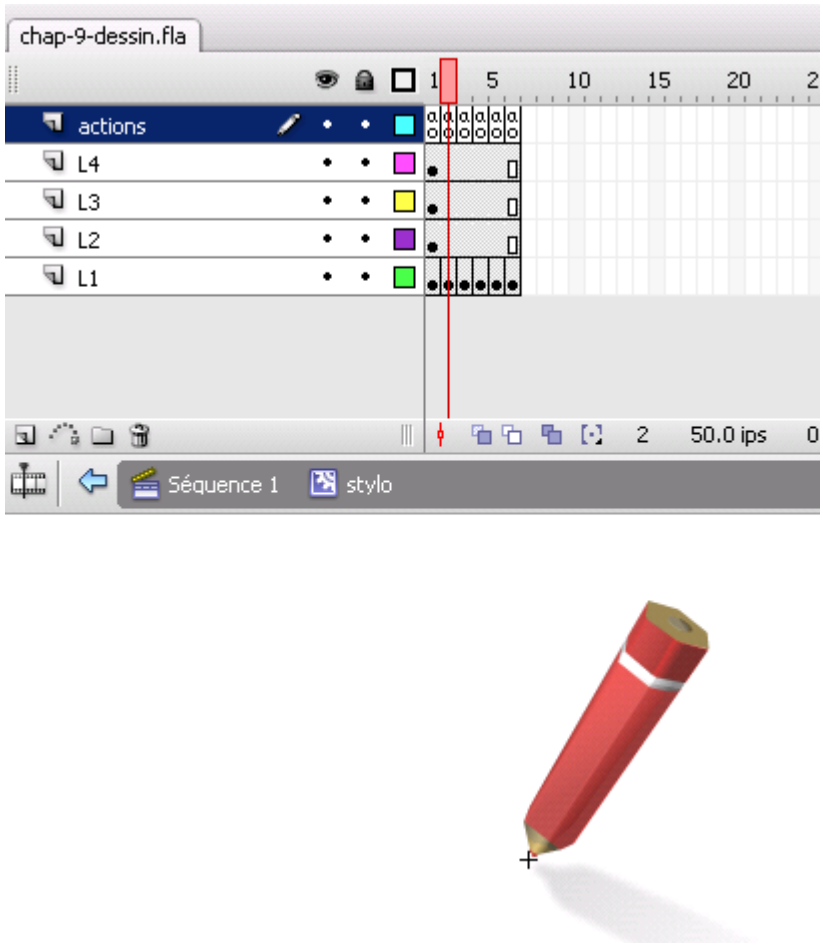

Figure 9-12. Ajout des différentes couleurs.

Lorsque l'utilisateur fait usage de la molette, nous déplaçons la tête de lecture du scénario du symbole. Nous ajoutons deux nouvelles propriétés qui nous permettront de positionner la tête de lecture lorsque la souris sera glissée :

```
// position de la tête de lecture
private var image:int;
private var index:int;
```

Nous initialisons au sein du constructeur la propriété `index` qui sera plus tard incrémentée, nous l'initialisons donc à 0 :

```
public function Stylo ()

{

     // le stylo écoute l'événement Event.ADDED_TO_STAGE
     // diffusé lorsque celui-ci est ajouté à la liste d'affichage
     addEventListener ( Event.ADDED_TO_STAGE, activation );

     // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
     // diffusé lorsque celui-ci est supprimé de la liste d'affichage
     addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );
```

```
            // initialisation des propriétés utilisées pour
            // le changement de couleurs
            image = 0;
            index = 1;

        }
```

Au sein de la méthode `activation` nous ajoutons un écouteur de l'événement `MouseEvent.MOUSE_WHEEL` auprès de l'objet `Stage` :

```
        private function activation ( pEvt:Event ):void

        {

            // cache le curseur
            Mouse.hide();

            // écoute des différents événements
            stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
            stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
            stage.addEventListener ( MouseEvent.MOUSE_UP, lacheSouris );
            stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );
            stage.addEventListener ( MouseEvent.MOUSE_WHEEL, moletteSouris );

        }
```

La méthode écouteur `moletteSouris` se charge de déplacer la tête de lecture sous forme de boucle, à l'aide de l'opérateur modulo `%`, la propriété `index` stocke l'index de la couleur sélectionnée :

```
        // déplace la tête de lecture lors de l'utilisation de la molette
        private function moletteSouris ( pEvt:MouseEvent ):void

        {

            gotoAndStop ( index = (++image%totalFrames)+1 );

        }
```

Si nous testons l'application, lorsque la molette de la souris est utilisée, le stylo change de couleur. Grâce au modulo, les couleurs défilent en boucle.

Il reste cependant à changer la couleur du tracé correspondant à la couleur du stylo choisie par l'utilisateur. Pour cela nous allons stocker toutes les couleurs disponibles dans un tableau au sein d'une propriété statique.

Pourquoi utiliser ici une propriété statique ?

Car les couleurs sont globales à toutes les occurrences du stylo qui pourraient être crées, ayant un sens global nous créons donc un tableau au sein d'une propriété `couleurs` statique :

```
        // tableau contenant les couleurs disponibles
        private static var couleurs:Array = [ 0x5BBA48, 0xEA312F, 0x00B7F1,
    0xFFF035, 0xD86EA3, 0xFBAE34 ];
```

Il ne nous reste plus qu'à modifier au sein de la méthode `clicSouris` la couleur du tracé, pour cela nous utilisons la propriété `index` qui représente l'index de la couleur au sein du tableau :

```
private function clicSouris ( pEvt:MouseEvent ):void

{

	if ( conteneur == null ) throw new Error ( "Veuillez appeler au
préalable la méthode affecteToile()" );

	// récupération des coordonnées de la souris en x et y
	positionX = pEvt.stageX;
	positionY = pEvt.stageY;

	// un nouvel objet Shape est crée pour chaque tracé
	monNouveauTrace = new Shape();

	// nous ajoutons le conteneur de tracé au conteneur principal
	conteneur.addChild ( monNouveauTrace );

	// puis nous référençons le tracé au sein du tableau
	// référençant les tracés affichés
	tableauTraces.push ( monNouveauTrace );

	// nous définissons un style de tracé
	monNouveauTrace.graphics.lineStyle ( 2, Stylo.couleurs[index-1], 1 );

	// la mine est déplacée à cette position
	// pour commencer à dessiner à partir de cette position
	monNouveauTrace.graphics.moveTo ( positionX, positionY );

	// si un nouveau tracé intervient alors que nous sommes
	// repartis en arrière nous repartons de cet état
	if ( tableauAncienTraces.length ) tableauAncienTraces = new Array;

	// écoute du mouvement de la souris
	stage.addEventListener ( MouseEvent.MOUSE_MOVE, dessine );

}
```

Lorsque la souris est enfoncée, nous pointons grâce à la propriété `index` au sein du tableau `couleurs` afin de choisir la couleur correspondante.

Nous ajoutons un petit détail final rendant notre classe plus souple, il serait intéressant de pouvoir en instanciant le stylo lui passer une vitesse permettant d'influencer sa vitesse de rotation.

Nous ajoutons une propriété friction de type `Number` :

```
// stocke la friction du stylo
private var friction:Number;
```

Puis nous modifions le constructeur est modifié afin d'accueillir la vitesse :

```
public function Stylo ( pFriction:Number=.1 )
```

```
        {

                // le stylo écoute l'événement Event.ADDED_TO_STAGE
                // diffusé lorsque celui-ci est ajouté à la liste d'affichage
                addEventListener ( Event.ADDED_TO_STAGE, activation );

                // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
                // diffusé lorsque celui-ci est supprimé de la liste d'affichage
                addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

                // initialisation des propriétés utilisées pour
                // le changement de couleurs
                image = 0;
                index = 1;

                // affecte la friction
                friction = pFriction;

        }
```

Enfin, nous modifions la méthode `bougeSouris` afin d'utiliser la valeur passée, stockée au sein de la propriété `friction` :

```
        private function bougeSouris ( pEvt:MouseEvent ):void

        {

                // récupération des coordonnées de la souris en x et y
                positionX = pEvt.stageX;
                positionY = pEvt.stageY;

                // affectation de la position
                x = positionX;
                y = positionY;

                // si le stylo passe dans la zone supérieure
                // alors nous inclinons le stylo
                if ( positionY < Stylo.LIMIT_Y )

                {

                        rotation -= ( rotation - ( Stylo.LIMIT_Y - positionY ) ) *
friction;

                // sinon, le stylo reprend son inclinaison d'origine
                } else rotation -= ( rotation - 0 ) * friction;

                // force le rafraîchissement
                pEvt.updateAfterEvent();

        }
```

Souvenons-nous d'un concept important de la programmation orientée objet : l'encapsulation !

Dans notre classe `Stylo` les propriétés sont toutes privées, car il n'y aucune raison que celles ci soient modifiables depuis l'extérieur. Bien entendu, il existe des situations dans lesquelles l'utilisation de propriétés privées n'a pas de sens.

Dans le cas d'une classe géometrique Point, il convient que les propriétés que les propriétés `x`, `y` et `z` doivent être publiques et accessibles. Il convient de cacher les propriétés qui ne sont pas utiles à l'utilisateur de la classe ou bien celles qui ont de fortes chances d'évoluer dans le temps.

> Souvenez-vous, l'intérêt est de pouvoir changer l'implémentation sans altérer l'interface de programmation.

En utilisant des propriétés privées au sein de notre classe `Stylo`, nous garantissons qu'aucun code extérieur à la classe ne peut accéder et ainsi altérer le fonctionnement de l'application.

La friction qui est passée à l'initialisation d'un stylo doit obligatoirement être comprise entre 0 exclu et 1. Si ce n'est pas le cas, l'inclinaison du stylo échoue et le développeur assiste au disfonctionnement de l'application.

Dans le code suivant le développeur ne connaît pas les valeurs possibles, et passe 10 comme vitesse d'inclinaison :

```
 // création du symbole
var monStylo:Stylo = new Stylo( 10 );
```

En testant l'application, le développeur se rend compte que le mécanisme d'inclinaison du stylo ne fonctionne plus. Cela est dû au fait que la valeur de friction exprime une force de frottement et doit être comprise entre 0 et 1.

N'oublions pas que nous sommes en train de développer un objet intelligent, autonome qui doit pouvoir évoluer dans différentes situations et pouvoir indiquer au développeur ce qui ne va pas.

Cela ne vous rappelle rien ?

Souvenez-vous d'un point essentiel que nous avons traité lors du précédent chapitre, *le contrôle d'affectation*.

Nous allons ajouter un test au sein du constructeur afin de vérifier si la valeur passée est acceptable, si ce n'est pas le cas nous passerons une valeur par défaut qui assurera le bon fonctionnement du stylo, à l'inverse si la valeur passée est incorrecte nous afficherons un message d'erreur.

Nous rajoutons la condition au sein du constructeur :

```
public function Stylo ( pFriction:Number=.1 )

{
```

```
            // le stylo écoute l'événement Event.ADDED_TO_STAGE
            // diffusé lorsque celui-ci est ajouté à la liste d'affichage
            addEventListener ( Event.ADDED_TO_STAGE, activation );

            // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
            // diffusé lorsque celui-ci est supprimé de la liste d'affichage
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

            // initialisation des propriétés utilisées pour
            // le changement de couleurs
            image = 0;
            index = 1;

            // affecte la friction
            if ( pFriction > 0 && pFriction <= 1 ) friction = pFriction;

            else

            {
                  trace("Erreur : Friction non correcte, la valeur doit être
comprise entre 0 et 1");
                  friction = .1;

            }

      }
```

Grâce à ce test, nous contrôlons l'affectation afin d'être sur de la bonne exécution de l'application. Cette fois le développeur a passé une valeur supérieure à 1, l'affectation est contrôlée, un message d'information indique ce qui ne va pas :

```
/* affiche :
Erreur : Friction non correcte la valeur doit être comprise entre 0 et 1
*/
var monStylo:Stylo = new Stylo( 50 );
```

De cette manière, le développeur tiers possède toutes les informations pour s'assurer du bon fonctionnement du stylo. Elégant n'est ce pas ?

Il faut cependant prévoir la possibilité de changer la vitesse du mouvement une fois l'objet crée, pour cela nous allons définir une méthode appelée `affecteVitesse` qui s'occupera d'affecter la propriété `friction`.

Afin de bien encapsuler notre classe nous allons contrôler l'affectation des propriétés grâce à une méthode spécifique :

```
      public function affecteVitesse ( pFriction:Number ):void

      {

            // affecte la friction
            if ( pFriction > 0 && pFriction <= 1 ) friction = pFriction;

            else

            {
```

```
            trace("Erreur : Friction non correcte, la valeur doit être
comprise entre 0 et 1");
            friction = .1;

      }

  }
```

Nous intégrons le code défini précédemment au sein du constructeur, afin de contrôler l'affectation à la propriété `friction`. Nous pouvons donc remplacer le code du constructeur par un appel à la méthode `affecteVitesse` :

```
  public function Stylo ( pFriction:Number=.1 )

  {

      // le stylo écoute l'événement Event.ADDED_TO_STAGE
      // diffusé lorsque celui-ci est ajouté à la liste d'affichage
      addEventListener ( Event.ADDED_TO_STAGE, activation );

      // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
      // diffusé lorsque celui-ci est supprimé de la liste d'affichage
      addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

      // initialisation des propriétés utilisées pour
      // le changement de couleurs
      image = 0;
      index = 1;

      affecteVitesse ( pFriction );

  }
```

En ajoutant une méthode `affecteVitesse`, l'affectation de la propriété `friction` est contrôlée.

Voici le code final de la classe `Stylo` :

```
package

{

  import flash.display.MovieClip;
  import flash.display.Shape;
  import flash.display.DisplayObjectContainer;
  import flash.events.MouseEvent;
  import flash.events.KeyboardEvent;
  import flash.events.Event;
  import flash.ui.Mouse;
  import flash.ui.Keyboard;

  public class Stylo extends MovieClip

  {

      // limite pour l'axe des y
      private static const LIMIT_Y:int = 140;

      // position en cours de la souris
```

```
        private var positionX:Number;
        private var positionY:Number;

        // stocke une référence au conteneur de dessin
        private var conteneur:DisplayObjectContainer;

        // tableaux référençant les formes tracées et supprimées
        private var tableauTraces:Array = new Array();
        private var tableauAncienTraces:Array = new Array();

        // référence le tracé en cours
        private var monNouveauTrace:Shape;

        // code des touches Y et Z
        private static const codeToucheY:int = 89;
        private static const codeToucheZ:int = 90;

        // position de la tête de lecture
        private var image:int;
        private var index:int;

        // tableau contenant les couleurs disponibles
        private static var couleurs:Array = [ 0x5BBA48, 0xEA312F, 0x00B7F1,
0xFFF035, 0xD86EA3, 0xFBAE34 ];

        // stocke la friction du stylo
        private var friction:Number;

        public function Stylo ( pFriction:Number=.1 )

        {

             // le stylo écoute l'événement Event.ADDED_TO_STAGE
             // diffusé lorsque celui-ci est ajouté à la liste d'affichage
             addEventListener ( Event.ADDED_TO_STAGE, activation );

             // le stylo écoute l'événement Event.REMOVED_FROM_STAGE
             // diffusé lorsque celui-ci est supprimé de la liste d'affichage
             addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

             // initialisation des propriétés utilisées pour
             // le changement de couleurs
             image = 0;
             index = 1;

             // affectation contrôlée de la vitesse
             affecteVitesse ( pFriction );

        }

        private function activation ( pEvt:Event ):void

        {

             // cache le curseur
             Mouse.hide();

             // écoute des différents événements
             stage.addEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
             stage.addEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
             stage.addEventListener ( MouseEvent.MOUSE_UP, relacheSouris );
             stage.addEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier );
```

```
            stage.addEventListener ( MouseEvent.MOUSE_WHEEL, moletteSouris );

      }

      private function desactivation ( pEvt:Event ):void

      {

            // affiche le curseur
            Mouse.show();

            // arrête l'écoute des différents événements
            stage.removeEventListener ( MouseEvent.MOUSE_MOVE, bougeSouris );
            stage.removeEventListener ( MouseEvent.MOUSE_DOWN, clicSouris );
            stage.removeEventListener ( MouseEvent.MOUSE_UP, relacheSouris );
            stage.removeEventListener ( KeyboardEvent.KEY_DOWN, ecouteClavier
);
            stage.removeEventListener ( MouseEvent.MOUSE_WHEEL, moletteSouris
);

      }

      // déplace la tête de lecture au scroll souris
      private function moletteSouris ( pEvt:MouseEvent ):void

      {

            gotoAndStop ( index = (++image%totalFrames)+1 );

      }

      private function ecouteClavier ( pEvt:KeyboardEvent ):void

      {

            // si la barre espace est enfoncée
            if ( pEvt.keyCode == Keyboard.SPACE )

            {

                  // nombre d'objets Shape contenant des tracés
                  var lng:int = tableauTraces.length;

                  // suppression des traçés de la liste d'affichage
                  while ( lng-- ) conteneur.removeChild ( tableauTraces[lng]
);

                  // les tableaux d'historiques sont reinitialisés
                  // les références supprimées
                  tableauTraces = new Array();
                  tableauAncienTraces = new Array();
                  monNouveauTrace = null;

            }

            if ( pEvt.ctrlKey )

            {

                  // si retour en arrière (CTRL+Z)
                  if( pEvt.keyCode == Stylo.codeToucheZ &&
tableauTraces.length )
```

```
                {

                    // nous supprimons le dernier tracé
                    var aSupprimer:Shape = tableauTraces.pop();

                    // nous stockons chaque tracé supprimé
                    // dans le tableau spécifique
                    tableauAncienTraces.push ( aSupprimer );

                    // nous supprimons le tracé de la liste d'affichage
                    conteneur.removeChild( aSupprimer );

                // si retour en avant (CTRL+Y)
                } else if ( pEvt.keyCode == Stylo.codeToucheY &&
tableauAncienTraces.length )

                {

                    // nous récupérons le dernier tracé ajouté
                    var aAfficher:Shape = tableauAncienTraces.pop();

                    // nous le replaçons dans le tableau de tracés à
l'affichage
                    tableauTraces.push ( aAfficher );

                    // puis nous l'affichons
                    conteneur.addChild ( aAfficher );

                }

            }

        }

        private function clicSouris ( pEvt:MouseEvent ):void

        {

            if ( conteneur == null ) throw new Error ( "Veuillez appeler au
préalable la méthode affecteToile()" );

            // récupération des coordonnées de la souris en x et y
            positionX = pEvt.stageX;
            positionY = pEvt.stageY;

            // un nouvel objet Shape est crée pour chaque tracé
            monNouveauTrace = new Shape();

            // nous ajoutons le conteneur de tracé au conteneur principal
            conteneur.addChild ( monNouveauTrace );

            // puis nous référençons le tracé au sein du tableau
            // référençant les tracés affichés
            tableauTraces.push ( monNouveauTrace );

            // nous définissons un style de tracé
            monNouveauTrace.graphics.lineStyle ( 2, Stylo.couleurs[index-1],
1 );

            // la mine est déplacée à cette position
            // pour commencer à dessiner à partir de cette position
```

```
            monNouveauTrace.graphics.moveTo ( positionX, positionY );

            // si un nouveau tracé intervient alors que nous sommes
            // repartis en arrière nous repartons de cet état
            if ( tableauAncienTraces.length ) tableauAncienTraces = new
Array;

            // écoute du mouvement de la souris
            stage.addEventListener ( MouseEvent.MOUSE_MOVE, dessine );

      }

      private function bougeSouris ( pEvt:MouseEvent ):void

      {

            // récupération des coordonnées de la souris en x et y
            positionX = pEvt.stageX;
            positionY = pEvt.stageY;

            // affectation de la position
            x = positionX;
            y = positionY;

            // si le stylo passe dans la zone supérieure
            // alors nous inclinons le stylo
            if ( positionY < Stylo.LIMIT_Y )

            {

                  rotation -= ( rotation - ( Stylo.LIMIT_Y - positionY ) ) *
friction;

            // sinon, le stylo reprend son inclinaison d'origine
            } else rotation -= ( rotation - 0 ) * friction;

            // force le rafraîchissement
            pEvt.updateAfterEvent();

      }

      private function relacheSouris ( pEvt:MouseEvent ):void

      {

            stage.removeEventListener ( MouseEvent.MOUSE_MOVE, dessine );

      }

      private function dessine ( pEvt:MouseEvent ):void
      {

            if ( monNouveauTrace != null )

            {

                  // la mine est déplaçée à cette position
                  // pour commencer à dessiner à partir de cette position
                  monNouveauTrace.graphics.lineTo ( positionX, positionY );

            }
```

```
        }

        // méthode permettant de spécifier le conteneur du dessin
        public function affecteToile ( pToile:DisplayObjectContainer ):void

        {

             conteneur = pToile;

        }

        public function affecteVitesse ( pFriction:Number ):void

        {

             // affecte la friction
             if ( pFriction > 0 && pFriction <= 1 ) friction = pFriction;

             else

             {
                  trace("Erreur : Friction non correcte, la valeur doit être
comprise entre 0 et 1");
                  friction = .1;

             }

        }

   }

}
```

A retenir

- Etendre les classes graphiques natives de Flash permet de créer des objets interactifs puissants et réutilisables.
- Au sein d'une sous-classe graphique, l'utilisation du mot clé `this` fait directement référence au scénario du symbole.

Réutiliser le code

Notre application de dessin plaît beaucoup, une agence vient de nous contacter afin de décliner l'application pour un nouveau client. Grâce à notre structure actuelle, la déclinaison graphique ne va nécessiter aucune modification du code.

Dans un nouveau document Flash CS3 nous importons un nouveau graphisme relatif au stylo comme l'illustre la figure 9-13 :

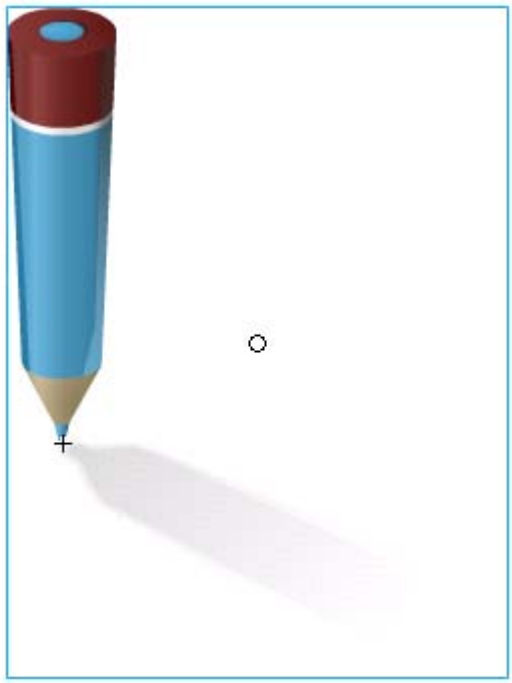

Figure 9-13. Nouveau graphisme.

Au sein du panneau *Propriétés de liaison* nous spécifions la classe `Stylo` développée précédemment.

Voilà, nous pouvons compiler, la figure 9-14 illustre le résultat :

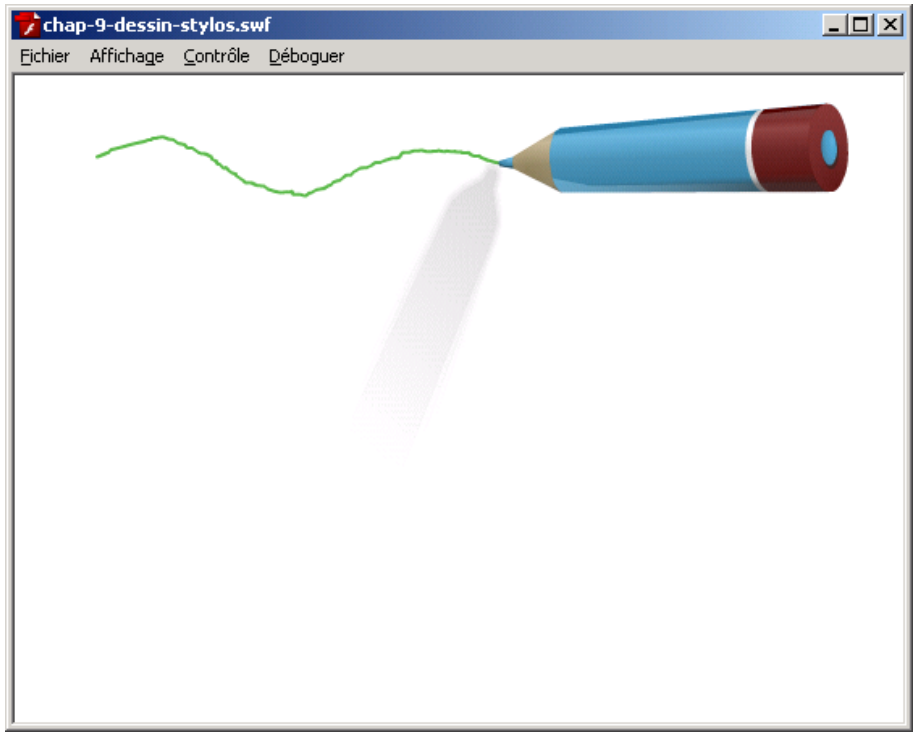

Figure 9-14. Réutilisation du code de la classe `Stylo`.

La déclinaison de notre projet a pris quelques secondes seulement, tout symbole de type `MovieClip` peut être lié à la classe `Stylo`, et bénéficier de toutes ses fonctionnalités et comportements.

Classe dynamique

Comme nous l'avons vu lors des précédents chapitres, il existe en ActionScript 3 deux types de classes : dynamiques et non dynamiques.

Dans le cas de sous-classes graphiques, quelque soit la nature de super-classe, la sous-classe graphique est par défaut toujours non dynamique. Il est donc impossible à ce stade d'ajouter une propriété ou une méthode à l'exécution à une instance de la classe `Stylo` :

```
// création du symbole
var monStylo:Stylo = new Stylo( .1 );

// ajout d'une propriété à l'exécution
monStylo.maProp = 12;
```

Le code précédent génère l'erreur suivante à la compilation :

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

Ce comportement par défaut est de bon augure, car il est généralement déconseillé d'avoir recours à des classes dynamiques. En donnant la possibilité de modifier l'implémentation à l'exécution, le développeur

utilisant la classe doit parcourir le code d'un projet afin de découvrir les comportements qui peuvent être ajoutés à l'exécution. En lisant la classe, ce dernier n'est pas renseigné de toutes les capacités et caractéristiques de celle-ci.

Dans de rares situations, il peut toutefois être nécessaire de rendre la sous-classe graphique dynamique, pour cela nous ajoutons l'attribut `dynamic` devant le mot clé `class` :

```
dynamic public class Stylo extends MovieClip
```

Une fois la classe modifiée, le code suivant fonctionne :

```
// création du symbole
var monStylo:Stylo = new Stylo( .1 );

// ajout d'une propriété à l'exécution
monStylo.maProp = 12;

// affiche : 12
trace( monStylo.maProp);
```

Bien que l'attribut `dynamic` existe, seule la classe `MovieClip` en profite en ActionScript 3, toutes les autres classes ne permettent pas l'ajout de propriétés ou méthodes à l'exécution et sont dites non dynamiques.

Il est fortement déconseillé de s'appuyer sur des classes dynamiques dans vos projets. En rendant une classe dynamique aucune vérification de type n'est faite à la compilation, nous pouvons donc même si celle-ci nous renvoie `undefined`, accéder à des propriétés privées :

```
// création du symbole
var monStylo:Stylo = new Stylo( .1 );

// aucune vérification de type à la compilation
// heureusement, la machine virtuelle (VM2) conserve les types
// à l'exécution et empêche son accès en renvoyant undefined
// affiche : undefined
trace( monStylo.nFriction );
```

La machine virtuelle 2 (VM2) conserve les types à l'exécution, ainsi lorsque nous accédons à une propriété privée au sein d'une classe dynamique, celle-ci renvoie `undefined`. Dans le cas d'une méthode privée, une erreur à l'exécution de type `TypeError` est levée.

A retenir

- L'attribut `dynamic` permet de rendre une classe dynamique.
- Aucune vérification de type n'est faite à la compilation sur une classe dynamique.
- Il est déconseillé d'utiliser des classes dynamiques dans des projets ActionScript.
- Dans le cas de sous-classes graphiques, quelque soit la nature de super-classe, la sous-classe graphique est par défaut toujours non dynamique.

Un vrai constructeur

En ActionScript 1 et 2, il était aussi possible d'étendre la classe `MovieClip`. Le principe était quasiment le même, un symbole était défini dans la librairie, puis une classe était liée au symbole.

Attention, il est important de noter qu'en ActionScript 2 la sous-classe graphique que nous pouvions définir était *liée au symbole*, cela signifie que seul l'appel de la méthode `attachMovie` instanciait la classe.

En ActionScript 3 grâce au nouveau modèle d'instanciation des objets graphiques, *c'est le symbole qui est lié à la classe*. Cela signifie que l'instanciation de la sous-classe entraîne l'affichage du symbole et non l'inverse.

Le seul moyen d'instancier notre sous-classe graphique en ActionScript 2 était d'appeler la méthode `attachMovie`. ActionScript 2, contrairement à ActionScript 3 souffrait d'un lourd héritage, ce processus d'instanciation des objets graphiques était archaïque et ne collait pas au modèle objet. En interne le lecteur déclenchait le constructeur de la sous-classe graphique. Il était impossible de passer des paramètres d'initialisation, seul l'objet d'initialisation (`initObject`) permettait de renseigner des propriétés avant même que le constructeur ne soit déclenché.

En instanciant la sous-classe, le symbole n'était pas affiché, le code suivant ne fonctionnait pas car le seul moyen d'afficher le clip pour le lecteur était la méthode `attachMovie` :

```
// la sous-classe était instanciée mais le symbole n'était pas affiché
var monSymbole:SousClasseMC = new SousClasseMC();
```

De plus, la liaison entre le symbole et la classe se faisait uniquement à travers le panneau *Propriétés de Liaison*, ainsi il était impossible de lier une classe à un objet graphique crée par programmation.

ActionScript 3 corrige cela. Dans l'exemple suivant nous allons créer une sous classe de `flash.display.Sprite` sans créer de symbole au sein de la bibliothèque, tout sera réalisé dynamiquement.

A retenir

- En ActionScript 2, il était impossible de passer des paramètres au constructeur d'une sous classe graphique.
- Afin de palier à ce problème, nous utilisions l'objet d'initialisation disponible au sein de la méthode `attachMovie`.
- ActionScript 3 règle tout cela, grâce au nouveau modèle d'instanciation des objets graphiques.
- En ActionScript 2, la classe était liée au symbole. Seule la méthode `attachMovie` permettait d'instancier la sous-classe graphique.
- En ActionScript 3, le symbole est lié à la classe. L'instanciation de la sous-classe par le mot clé `new` entraîne la création du symbole.

Créer des boutons dynamiques

Nous allons étendre la classe `Sprite` afin de créer des boutons dynamiques. Ces derniers nous permettront de créer un menu qui pourra être modifié plus tard afin de donner vie à d'autres types de menus.

Lors du chapitre 7 nous avions développé un menu composé de boutons de type `Sprite`. Tous les comportements étaient définis à l'extérieur de celui-ci, nous devions d'abord créer le symbole correspondant, puis au sein de la boucle ajouter les comportements boutons, les différents effets de survol, puis ajouter le texte. Il était donc impossible de recréer rapidement un bouton identique au sein d'une autre application.

En utilisant une approche orientée objet à l'aide d'une sous-classe graphique, nous allons pouvoir définir une classe `Bouton` qui pourra être réutilisée dans chaque application nécessitant un bouton fonctionnel.

Nous allons à présent organiser nos classes en les plaçant dans des paquetages spécifiques, au cours du chapitre précédent nous avons traité la notion de paquetages sans véritablement l'utiliser. Il est temps d'utiliser cette notion, cela va nous permettre d'organiser nos classes proprement et éviter dans certaines situations les conflits entre les classes.

Nous créons un nouveau document Flash CS3, à coté de celui-ci nous créons un répertoire `org` contenant un répertoire `bytearray.` Puis

au sein même de ce répertoire nous créons un autre répertoire appelé `ui`.

Au sein du répertoire `ui` nous créons une sous-classe de `Sprite` appelée `Bouton` contenant le code suivant :

```
package org.bytearray.ui

{
  import flash.display.Sprite;

  public class Bouton extends Sprite
  {

        public function Bouton ()

        {

             trace ( this );

        }

  }


}
```

Nous remarquons que le paquetage reflète l'emplacement de la classe au sein des répertoires, si le chemin n'est pas correct le compilateur ne trouve pas la classe et génère une erreur. Pour instancier la classe `Bouton` dans notre document Flash nous devons obligatoirement l'importer, car le compilateur recherche par défaut les classes situées à coté du fichier .fla mais ne parcours pas tous les répertoires voisins afin de trouver la définition de classe.

Nous lui indiquons où elle se trouve à l'aide du mot clé `import` :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// affiche : [object Bouton]
var monBouton:Bouton = new Bouton();
```

Notre bouton est bien crée mais pour l'instant cela ne nous apporte pas beaucoup plus qu'une instanciation directe de la classe `Sprite`, nous allons tout de suite ajouter quelques fonctionnalités. Nous allons dessiner le bouton dynamiquement à l'aide de l'API de dessin. Grâce à l'héritage, la classe `Bouton` possède toutes les capacités d'un `Sprite` et hérite donc d'une propriété `graphics` propre à l'API de dessin.

Nous pourrions dessiner directement au sein de la classe `Bouton` mais nous préfèrerons l'utilisation d'un objet `Shape` dédié à cela. Si nous devons plus tard ajouter un effet de survol nous déformerons celui-ci

afin de ne pas étirer l'enveloppe principale du bouton qui provoquerait un étirement global de tous les enfants du bouton.

Au sein du constructeur de la classe `Bouton` nous dessinons le bouton :

```
package org.bytearray.ui

{
  import flash.display.Shape;
  import flash.display.Sprite;

  public class Bouton extends Sprite

  {

        // référence le fond du bouton
        private var fondBouton:Shape;

        public function Bouton ()

        {

              // création du fond du bouton
              fondBouton = new Shape();

              // ajout à la liste d'affichage
              addChild ( fondBouton );

              // dessine le bouton
              fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
              fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

        }

  }

}
```

Nous choisissons une couleur aléatoire, puis nous dessinons une forme rectangulaire, plus tard nous pourrons choisir la couleur du bouton lors de sa création ou encore choisir sa dimension.

Il ne nous reste plus qu'à l'afficher :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

// instanciation
// affiche : [object Bouton]
var monBouton:Bouton = new Bouton();

// ajout au sein du conteneur
conteneurMenu.addChild ( monBouton );

// affichage des boutons
addChild ( conteneurMenu );
```

Nous obtenons le résultat illustré en figure 9-15 :

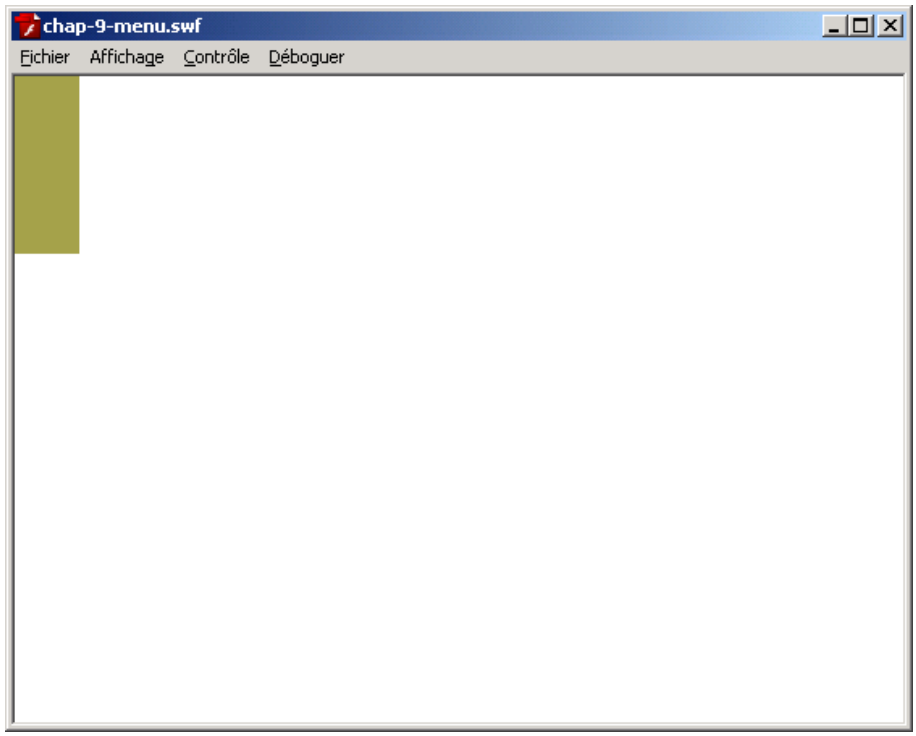

Figure 9-15. Instance de la classe `Bouton`.

Afin de rendre notre bouton cliquable, nous devons activer une propriété ?

Vous souvenez-vous de laquelle ?

La propriété `buttonMode` permet d'activer le comportement bouton auprès de n'importe quel `DisplayObject` :

```
package org.bytearray.ui

{
  import flash.display.Shape;
  import flash.display.Sprite;

  public class Bouton extends Sprite

  {

        // stocke le fond du bouton
        private var fondBouton:Shape;

        public function Bouton ()

        {

              // création du fond du bouton
              fondBouton = new Shape();
```

```
            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // dessine le bouton
            fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
            fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

            // activation du mode bouton
            buttonMode = true;

        }

    }

}
```

Le curseur main s'affiche alors au survol du bouton :

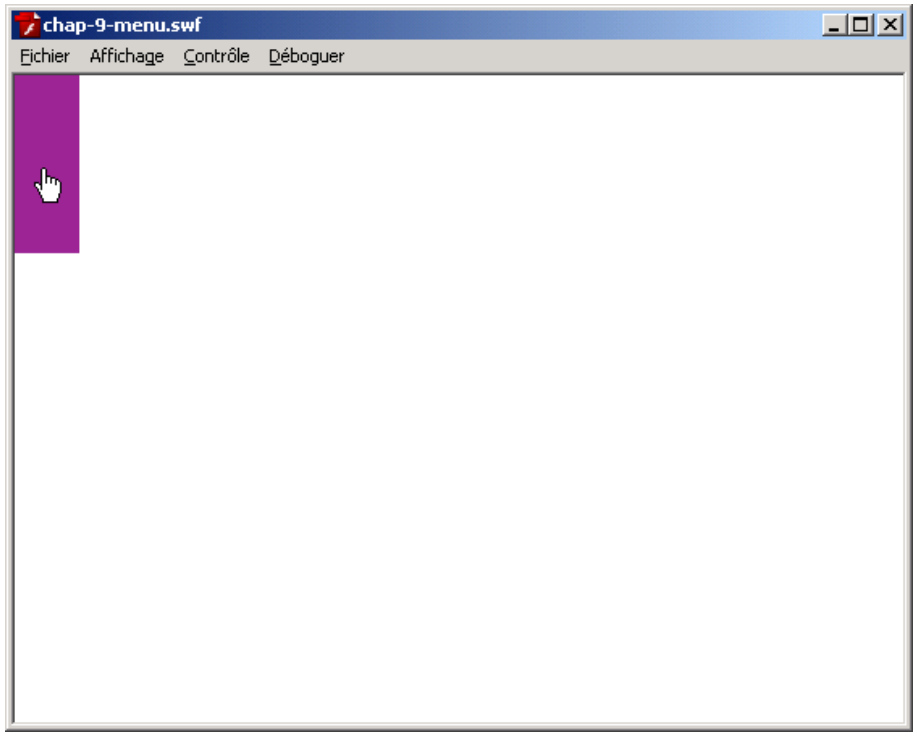

Figure 9-16. Activation du mode bouton.

Nous allons donner un peu de mouvement à ce dernier, en utilisant la classe `Tween` déjà abordée lors du chapitre 7.

Nous importons la classe `Tween` puis nous créons un objet du même type afin de gérer le survol :

```
package org.bytearray.ui

{

    import flash.display.Shape;
    import flash.display.Sprite;
    // import des classes Tween liées au mouvement
```

```
  import fl.transitions.Tween;
  import fl.transitions.easing.Bounce;

  public class Bouton extends Sprite

  {

       // stocke le fond du bouton
       private var fondBouton:Shape;
       // stocke l'objet Tween pour les différents état du bouton
       private var interpolation:Tween;

       public function Bouton ()

       {

            // création du fond du bouton
            fondBouton = new Shape();

            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // dessine le bouton
            fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
            fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

            // activation du mode bouton
            buttonMode = true;

            // création de l'objet Tween
            interpolation = new Tween (fondBouton, "scaleX", Bounce.easeOut,
  1, 1, 1, true );

       }

  }

}
```

Nous demandons à l'objet `Tween` de s'occuper de la propriété `scaleX` pour donner un effet d'étirement à l'objet `Shape` servant de fond à notre bouton. Nous allons donner un effet de rebond exprimé par la classe `Bounce`.

Lorsque le bouton est survolé nous démarrons l'animation. Nous écoutons l'événement `MouseEvent.ROLL_OVER` :

```
package org.bytearray.ui

{

  import flash.display.Shape;
  import flash.display.Sprite;
  // import des classes Tween liées au mouvement
  import fl.transitions.Tween;
  import fl.transitions.easing.Bounce;
  // import de la classe MouseEvent
  import flash.events.MouseEvent;

  public class Bouton extends Sprite
```

```
 {

      // stocke le fond du bouton
      private var fondBouton:Shape;
      // stocke l'objet Tween pour les différents état du bouton
      private var interpolation:Tween;

      public function Bouton ()

      {

            // création du fond du bouton
            fondBouton = new Shape();

            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // dessine le bouton
            fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
            fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

            // activation du mode bouton
            buttonMode = true;

            // création de l'objet Tween
            interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, 1, true );

            // écoute de l'événement MouseEvent.CLICK
            addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

      }

      // déclenché lors du survol du bouton
      private function survolSouris ( pEvt:MouseEvent ):void

      {

            // démarrage de l'animation
            interpolation.continueTo ( 2, 2 );

      }

 }

}
```

Lors du survol, le bouton s'étire avec un effet de rebond, la figure 9-17 illustre l'effet :

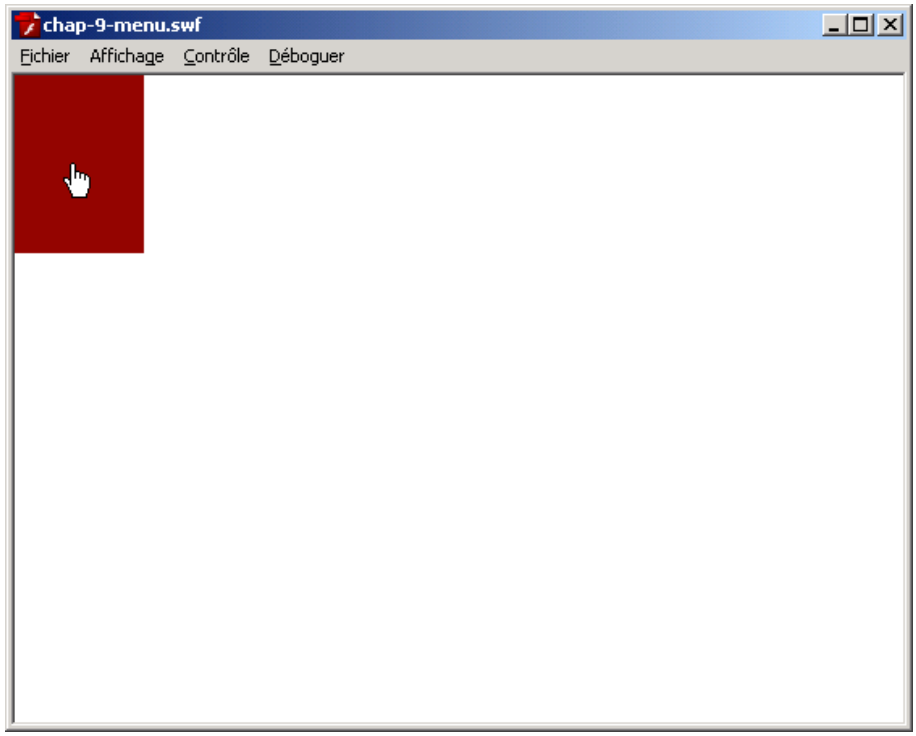

Figure 9-17. Etirement du bouton.

En ajoutant d'autres boutons nous obtenons un premier menu :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

var lng:int = 5;

var monBouton:Bouton;

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

{

  // instanciation
  // création de boutons rouge
  monBouton = new Bouton();

  //positionnement
  monBouton.y = 50 * i;

  // ajout au sein du conteneur
  conteneurMenu.addChild ( monBouton );

}

// affichage des boutons
addChild ( conteneurMenu );
```

Le résultat est illustré en figure 9-18 :

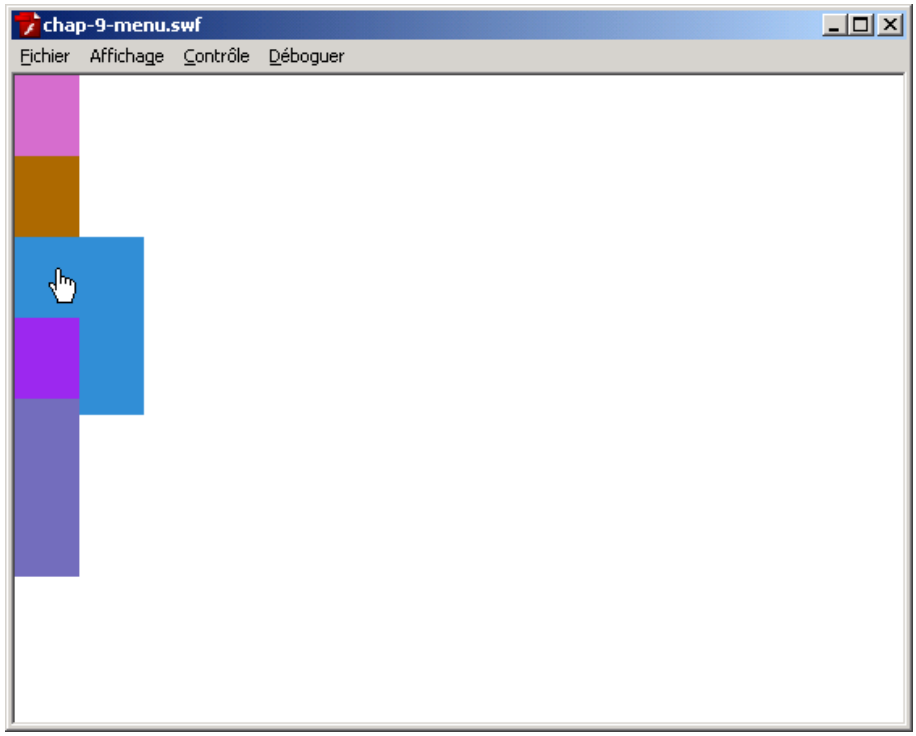

Figure 9-18. Création du menu.

La classe `Bouton` peut ainsi être réutilisée dans n'importe quel projet nécessitant un seul bouton, ou un menu.

Il manque pour le moment une fonctionnalité permettant de refermer le bouton cliqué lorsqu'un autre est sélectionné. Pour cela nous devons obligatoirement posséder une référence envers tous les boutons du menu, et pouvoir décider quels boutons refermer.

Vous souvenez-vous de la classe `Joueur` crée au cours du chapitre 8 ?

Nous avions défini un tableau stockant les références de chaque joueur crée, cela nous permettait à tout moment de savoir combien de joueurs étaient crées et de pouvoir y accéder. Nous allons reproduire le même mécanisme à l'aide d'un tableau statique.

Le tableau `tableauBoutons` stocke chaque référence de boutons :

```
package org.bytearray.ui

{

  import flash.display.Shape;
  import flash.display.Sprite;
  // import des classes Tween liées au mouvement
  import fl.transitions.Tween;
  import fl.transitions.easing.Bounce;
  // import de la classe MouseEvent
  import flash.events.MouseEvent;
```

```
public class Bouton extends Sprite

{

      // stocke le fond du bouton
      private var fondBouton:Shape;
      // stocke l'objet Tween pour les différents état du bouton
      private var interpolation:Tween;
      // stocke les références aux boutons
      private static var tableauBoutons:Array = new Array();

      public function Bouton ()

      {

            // ajoute chaque instance au tableau
            Bouton.tableauBoutons.push ( this );

            // création du fond du bouton
            fondBouton = new Shape();

            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // dessine le bouton
            fondBouton.graphics.beginFill ( Math.random()*0xFFFFFF, 1 );
            fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

            // activation du mode bouton
            buttonMode = true;

            // création de l'objet Tween
            interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, 1, true );

            // écoute de l'événement MouseEvent.CLICK
            addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

      }

      // déclenché lors du survol du bouton
      private function survolSouris ( pEvt:MouseEvent ):void

      {

            // démarrage de l'animation
            interpolation.continueTo ( 2, 2 );

      }

}

}
```

A tout moment un bouton peut parcourir le tableau statique et accéder à ses frères et décider de les refermer.

Nous modifions la méthode `survolSouris` afin d'appeler sur chaque bouton la méthode `fermer` :

```
// déclenché lors du survol du bouton
private function survolSouris ( pEvt:MouseEvent ):void

{

      // stocke la longueur du tableau
      var lng:int = Bouton.tableauBoutons.length;

      for (var i:int = 0; i<lng; i++ ) Bouton.tableauBoutons[i].fermer();

      // démarrage de l'animation
      interpolation.continueTo ( 2, 1 );

}
```

La méthode `fermer` est privée car celle ne sera appelée qu'un sein de la classe `Bouton` :

```
// méthode permettant de refermer le bouton
private function fermer  ():void

{
      // referme le bouton
      interpolation.continueTo ( 1, 1 );

}
```

Si nous testons l'animation nous remarquons que lors du survol les autres boutons ouverts se referment.

Bien entendu dans la plupart des projets ActionScript nous n'allons pas seulement créer des menus constitués de couleurs aléatoires. Il serait intéressant de pouvoir choisir la couleur de chaque bouton, pour cela nous ajoutons un paramètre afin de recevoir la couleur du bouton au sein du constructeur :

```
package org.bytearray.ui

{

  import flash.display.Shape;
  import flash.display.Sprite;
  // import des classes Tween liées au mouvement
  import fl.transitions.Tween;
  import fl.transitions.easing.Bounce;
  // import de la classe MouseEvent
  import flash.events.MouseEvent;

  public class Bouton extends Sprite

  {

      // stocke le fond du bouton
      private var fondBouton:Shape;
      // stocke l'objet Tween pour les différents état du bouton
      private var interpolation:Tween;
      // stocke les références aux boutons
      private static var tableauBoutons:Array = new Array();
      // stocke la couleur en cours du bouton
      private var couleur:Number;
```

```
        public function Bouton ( pCouleur:Number )

        {

             // ajoute chaque instance au tableau
             Bouton.tableauBoutons.push ( this );

             // création du fond du bouton
             fondBouton = new Shape();

             // ajout à la liste d'affichage
             addChild ( fondBouton );

             // stocke la couleur passée en paramètre
             couleur = pCouleur;

             // dessine le bouton
             fondBouton.graphics.beginFill ( couleur, 1 );
             fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

             // activation du mode bouton
             buttonMode = true;

             // création de l'objet Tween
             interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, 1, true );

             // écoute de l'événement MouseEvent.CLICK
             addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

        }

        // déclenché lors du survol du bouton
        private function survolSouris ( pEvt:MouseEvent ):void

        {
             // stocke la longueur du tableau
             var lng:int = Bouton.tableauBoutons.length;

             for (var i:int = 0; i<lng; i++ )
Bouton.tableauBoutons[i].fermer();

             // démarrage de l'animation
             interpolation.continueTo ( 2, 2 );

        }

        // méthode permettant de refermer le bouton
        private function fermer  ():void

        {
             // referme le bouton
             interpolation.continueTo ( 1, 1 );

        }

   }

}
```

Désormais la classe `Bouton` accepte un paramètre pour la couleur, le code suivant crée un menu constitué de boutons rouges seulement :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

var lng:int = 5;

var monBouton:Bouton;

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

{

  // instanciation
  // création de boutons rouge
  monBouton = new Bouton( 0x990000 );

  //positionnement
  monBouton.y = 50 * i;

  // ajout au sein du conteneur
  conteneurMenu.addChild ( monBouton );

}

// affichage des boutons
addChild ( conteneurMenu );
```

Cela n'a pas grand intérêt, nous allons donc modifier le code actuel afin d'associer à chaque bouton, une couleur précise. Pour cela nous stockons les couleurs au sein d'un tableau qui sera parcouru, le nombre de boutons du menu sera lié à la longueur du tableau :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

// tableau de couleurs
var couleurs:Array = [0x999900, 0x881122, 0x995471, 0x332100, 0x977821];

// nombre de couleurs
var lng:int = couleurs.length;

var monBouton:Bouton;

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

{

  // instanciation
  // création de boutons rouge
  monBouton = new Bouton( couleurs[i] );
```

```
        //positionnement
        monBouton.y = 50 * i;

        // ajout au sein du conteneur
        conteneurMenu.addChild ( monBouton );

      }

      // affichage des boutons
      addChild ( conteneurMenu );
```

Nous obtenons le menu illustré en figure 9-19 :

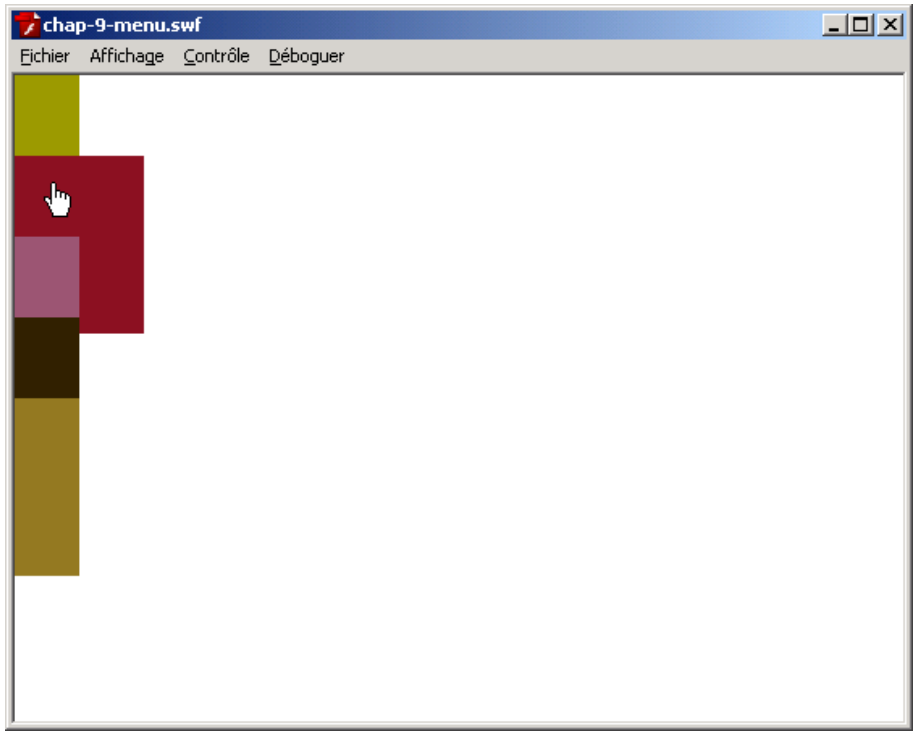

Figure 9-19. Menu constitué de couleurs prédéfinies.

Chaque bouton accepte une couleur lors de sa création, nous pourrions imaginer une application dans laquelle l'utilisateur choisit la couleur des boutons de son menu au sein d'une administration. Les couleurs sont alors récupérées d'une base de données puis utilisées pour créer ce menu. Nous verrons au cours du chapitre 19 intitulé *Flash Remoting* comment mettre en place cette application.

Notre menu n'est pas encore complet il serait judicieux de pouvoir spécifier la vitesse d'ouverture de chaque bouton. Souvenez-vous nous avons déjà intégré ce contrôle pour le mouvement du stylo de l'application précédente. Nos objets sont tous de la même famille mais possèdent des caractéristiques différentes, comme la couleur ou bien la vitesse d'ouverture.

Pour cela nous allons définir une propriété privée `vitesse` qui se chargera de stocker la vitesse passée à chaque bouton :

```
// stocke la vitesse d'ouverture de chaque bouton
private var vitesse:Number;
```

Puis nous définissons une méthode `affecteVitesse` qui se charge d'affecter la vitesse de manière contrôlée :

```
// gère l'affectation de la vitesse
public function affecteVitesse ( pVitesse:Number ):void

{

      // affecte la vitesse
      if ( pVitesse >= 1 && pVitesse <= 10 ) vitesse = pVitesse;

      else

      {
            trace("Erreur : Vitesse non correcte, la valeur doit être
comprise entre 1 et 10");
            vitesse = 1;

      }

}
```

Nous modifions le constructeur en ajoutant un paramètre appelé `pFriction` afin d'appeler la méthode `affecteVitesse` avant la création de l'objet `Tween` pour initialiser la propriété `vitesse` :

```
package org.bytearray.ui

{

  import flash.display.Shape;
  import flash.display.Sprite;
  // import des classes Tween liées au mouvement
  import fl.transitions.Tween;
  import fl.transitions.easing.Bounce;
  // import de la classe MouseEvent
  import flash.events.MouseEvent;

  public class Bouton extends Sprite

  {

      // stocke le fond du bouton
      private var fondBouton:Shape;
      // stocke l'objet Tween pour les différents état du bouton
      private var interpolation:Tween;
      // stocke les références aux boutons
      private static var tableauBoutons:Array = new Array();
      // stocke la couleur en cours du bouton
      private var couleur:Number;
      // stocke la vitesse d'ouverture de chaque bouton
      private var vitesse:Number;

      public function Bouton ( pCouleur:Number, pVitesse:Number=.1 )
```

```
        {

            // ajoute chaque instance au tableau
            Bouton.tableauBoutons.push ( this );

            // création du fond du bouton
            fondBouton = new Shape();

            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // stocke la couleur passée en paramètre
            couleur = pCouleur;

        // dessine le bouton
        fondBouton.graphics.beginFill ( couleur, 1 );
        fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

        // activation du mode bouton
        buttonMode = true;

        // affectation de la vitesse contrôlée
        affecteVitesse ( pVitesse );

        // création de l'objet Tween
        interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

        // écoute de l'événement MouseEvent.CLICK
        addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

    }

    // déclenché lors du survol du bouton
    private function survolSouris ( pEvt:MouseEvent ):void

    {

        // stocke la longueur du tableau
        var lng:int = Bouton.tableauBoutons.length;

        for (var i:int = 0; i<lng; i++ ) Bouton.tableauBoutons[i].fermer();

        // démarrage de l'animation
        interpolation.continueTo ( 2, vitesse );

    }

    // méthode permettant de refermer le bouton
    private function fermer  ():void

    {

        // referme le bouton
        interpolation.continueTo ( 1, vitesse );

    }

    // gère l'affectation de la vitesse
    public function affecteVitesse ( pVitesse:Number ):void
```

```
  {

        // affecte la vitesse
        if ( pVitesse >= 1 && pVitesse <= 10 ) vitesse = pVitesse;

        else

        {
              trace("Erreur : Vitesse non correcte, la valeur doit être
comprise entre 1 et 10");
              vitesse = 1;

        }

  }

  }

}
```

Dans le code suivant, nous créons un menu composé de boutons dont nous pouvons choisir la vitesse d'ouverture :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

// tableau de couleurs
var couleurs:Array = [0x999900, 0x881122, 0x995471, 0x332100, 0x977821];

// nombre de couleurs
var lng:int = couleurs.length;

var monBouton:Bouton;

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

{

  // instanciation
  // création de boutons rouge
  monBouton = new Bouton( couleurs[i], 1 );

  //positionnement
  monBouton.y = 50 * i;

  // ajout au sein du conteneur
  conteneurMenu.addChild ( monBouton );

}

// affichage du menu
addChild ( conteneurMenu );
```

Au cas où la vitesse passée ne serait pas correcte, le menu continue de fonctionner et un message d'erreur est affiché :

```
  // affiche : Erreur : Vitesse non correcte, la valeur
  // doit être comprise entre 1 et 10
  var monBouton:Bouton = new Bouton( couleurs[i], 0 );
```

Généralement, les boutons d'un menu contiennent une légende, nous allons donc ajouter un champ texte à chaque bouton. Pour cela nous importons les classes `flash.text.TextField` et `flash.text.TextFieldAutoSize` :

```
// import de la classe TextField et TextFieldAutoSize
import flash.text.TextField;
import flash.text.TextFieldAutoSize;
```

Nous créons une propriété `legende` afin de référencer la légende :

```
// légende du bouton
private var legende:TextField;
```

Puis nous ajoutons le champ au bouton au sein du constructeur :

```
public function Bouton ( pCouleur:Number, pVitesse:Number=.1 )

{

     // ajoute chaque instance au tableau
     Bouton.tableauBoutons.push ( this );

     // création du fond du bouton
     fondBouton = new Shape();

     // ajout à la liste d'affichage
     addChild ( fondBouton );

     // crée le champ texte
     legende = new TextField();

     // redimensionnement automatique du champ texte
     legende.autoSize = TextFieldAutoSize.LEFT;

     // rend le champ texte non sélectionnable
     legende.selectable = false;

     // ajout à la liste d'affichage
     addChild ( legende );

     // stocke la couleur passée en paramètre
     couleur = pCouleur;

     // dessine le bouton
     fondBouton.graphics.beginFill ( couleur, 1 );
     fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

     // activation du mode bouton
     buttonMode = true;

     // affectation de la vitesse contrôlée
     affecteVitesse ( pVitesse );

     // création de l'objet Tween
     interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

     // écoute de l'événement MouseEvent.CLICK
     addEventListener ( MouseEvent.ROLL_OVER, survolSouris );
```

```
}
```

Si nous testons notre menu, nous remarquons que le champ texte imbriqué dans le bouton reçoit les entrées souris et entre en conflit avec l'enveloppe principale du bouton.

Afin de désactiver les objets enfants du bouton nous passons la valeur `false` à la propriété `mouseChildren` du bouton :

```
// désactivation des objets enfants
mouseChildren = false;
```

Souvenez-vous, lors du chapitre 7, nous avons vu que la propriété `mouseChildren` permettait de désactiver les événements souris auprès des objets enfants.

Nous ajoutons un paramètre au constructeur de la classe `Bouton` afin d'accueillir le texte affiché par la légende :

```
  public function Bouton ( pCouleur:Number, pVitesse:Number=.1,
pLegende:String="Légende" )

  {

        // ajoute chaque instance au tableau
        Bouton.tableauBoutons.push ( this );

        // création du fond du bouton
        fondBouton = new Shape();

        // ajout à la liste d'affichage
        addChild ( fondBouton );

        // crée le champ texte
        legende = new TextField();

        // redimensionnement automatique du champ texte
        legende.autoSize = TextFieldAutoSize.LEFT;

        // rend le champ texte non sélectionnable
        legende.selectable = false;

        // ajout à la liste d'affichage
        addChild ( legende );

        // affecte la légende
        legende.text = pLegende;

        // stocke la couleur passée en paramètre
        couleur = pCouleur;

        // dessine le bouton
        fondBouton.graphics.beginFill ( couleur, 1 );
        fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

        // activation du mode bouton
        buttonMode = true;

        // désactivation des objets enfants
```

```
        mouseChildren = false;

        // affectation de la vitesse contrôlée
        affecteVitesse ( pVitesse );

        // création de l'objet Tween
        interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
1, vitesse, true );

        // écoute de l'événement MouseEvent.CLICK
        addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

    }
```

Les données utilisées afin de générer un menu proviennent généralement d'un flux XML ou de *Flash Remoting* et sont très souvent formatées sous forme de tableau associatif.

Nous allons réorganiser nos données afin d'utiliser un tableau associatif plutôt que plusieurs tableaux séparés :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

// tableau associatif contenant les données
var donnees:Array = new Array();

donnees.push ( { legende : "Accueil", vitesse : 1, couleur : 0x999900 } );
donnees.push ( { legende : "Photos", vitesse : 1, couleur : 0x881122 } );
donnees.push ( { legende : "Blog", vitesse : 1, couleur : 0x995471 } );
donnees.push ( { legende : "Liens", vitesse : 1, couleur : 0x332100 } );
donnees.push ( { legende : "Forum", vitesse : 1, couleur : 0x977821 } );

// nombre de rubriques
var lng:int = donnees.length;

var monBouton:Bouton;
var legende:String;
var couleur:Number;
var vitesse:Number;

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

{

  // récupération des infos
  legende = donnees[i].legende;
  couleur = donnees[i].couleur;
  vitesse = donnees[i].vitesse;

  // création des boutons
  monBouton = new Bouton( couleur, vitesse, legende );

  // positionnement
  monBouton.y = 50 * i;

  // ajout au sein du conteneur
  conteneurMenu.addChild ( monBouton );
```

```
}

// affichage du menu
addChild ( conteneurMenu );
```

En testant notre menu, nous obtenons le résultat illustré en figure 9-20 :

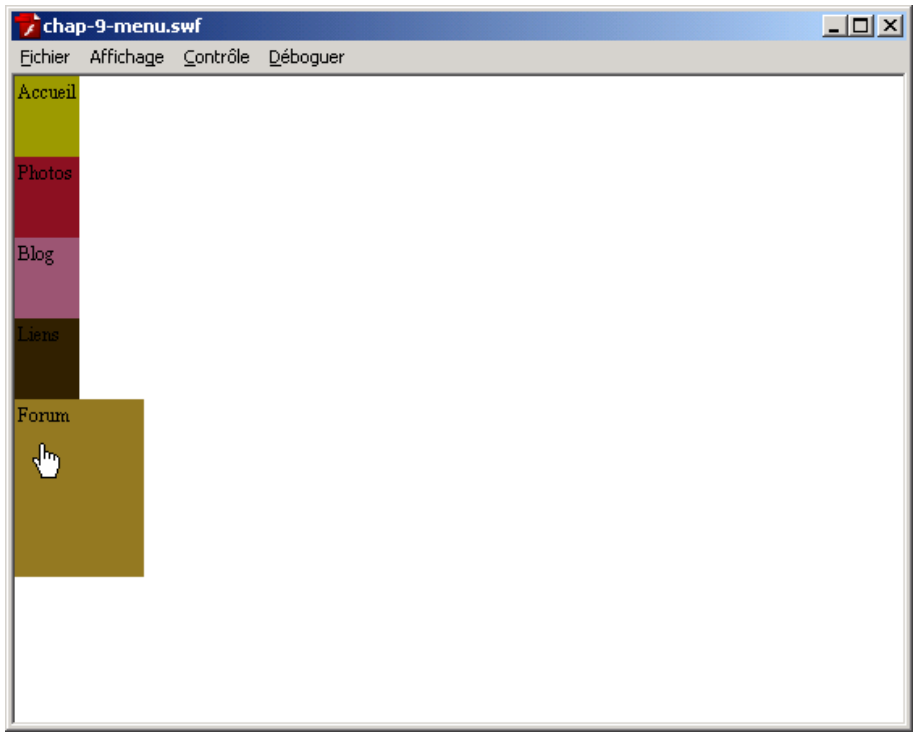

Figure 9-20. Boutons avec légendes.

Le texte de légende n'est pas correctement formaté, pour y remédier nous utilisons la classe `flash.text.TextFormat` ainsi que la classe `flash.text.Font`. Nous reviendrons plus en profondeur sur le formatage du texte au cours du chapitre 16 intitulé *Le texte*.

Afin d'intégrer une police au sein de la bibliothèque nous cliquons sur l'icône prévue illustrée en figure 9-21 :

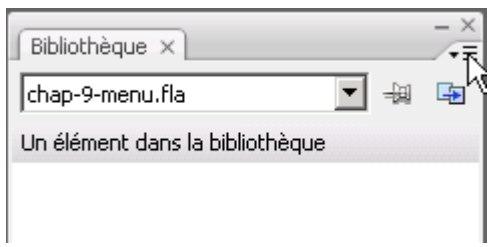

Figure 9-21. Options de la bibliothèque.

Le panneau d'options de bibliothèque s'ouvre, nous sélectionnons *Nouvelle Police* comme l'illustre la figure 9-22 :

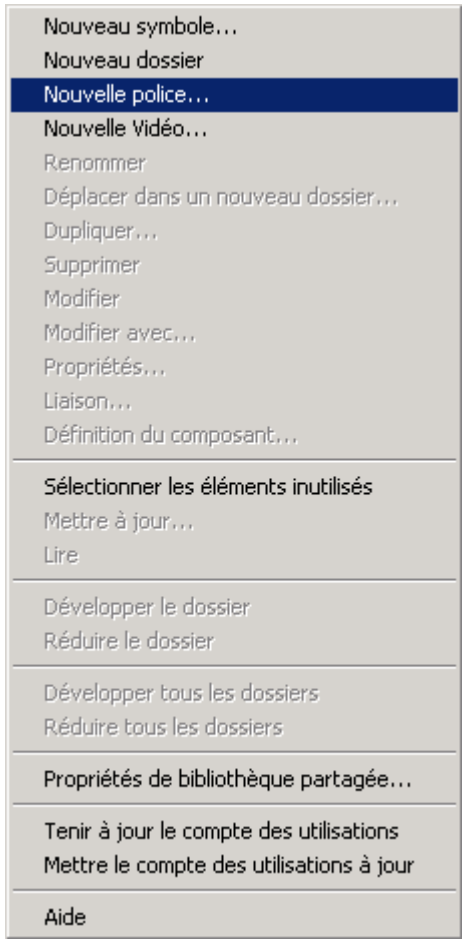

Figure 9-22. Insertion de police dans la bibliothèque.

Une fois sélectionnée, le panneau *Propriétés des symboles de police* s'affiche, ce dernier permet de sélectionner la police qui sera embarquée dans l'animation. Nous sélectionnons pour notre exemple la police *Trebuchet MS*, puis nous donnons un nom de classe à la police, nous conservons `Police 1`.

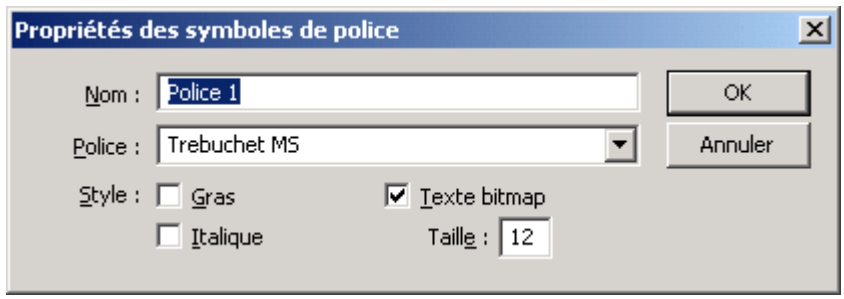

Figure 9-23. Propriétés des symboles de police.

Une fois validé, la police apparaît dans la bibliothèque, il ne nous reste plus qu'à l'utiliser au sein de nos champs texte contenus dans chaque bouton. Pour cela il faut lier cette police comme pour un symbole classique. En faisant un clic droit sur celle-ci, nous sélectionnons l'option *Liaisons* le panneau *Propriétés de liaison* s'ouvre comme l'illustre la figure 9-24 :

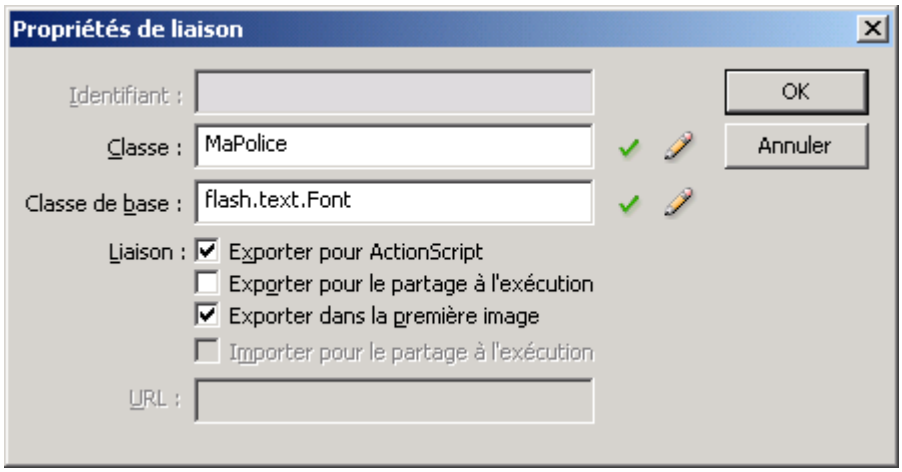

Figure 9-24. Panneau propriétés de liaison.

Nous choisissons `MaPolice` comme nom de classe, celle-ci va automatiquement être crée par Flash et héritera de la classe `flash.text.Font`.

Nous définissons une nouvelle propriété `formatage` afin de stocker l'objet `TextFormat` :

```
// formatage des légendes
private var formatage:TextFormat;
```

Nous importons la classe `flash.text.TextFormat` :

```
import flash.text.TextFormat;
```

Puis nous modifions le constructeur de la classe `Bouton` afin d'affecter le formatage et indiquer au champ texte d'utiliser la police embarquée :

```
public function Bouton ( pCouleur:Number, pVitesse:Number, pLegende:String
)

{

      // ajoute chaque instance au tableau
      Bouton.tableauBoutons.push ( this );

      // création du fond du bouton
      fondBouton = new Shape();

      // ajout à la liste d'affichage
      addChild ( fondBouton );
```

```
            // crée le champ texte
            legende = new TextField();

            // redimensionnement automatique du champ texte
            legende.autoSize = TextFieldAutoSize.LEFT;

            // ajout à la liste d'affichage
            addChild ( legende );

            // affecte la légende
            legende.text = pLegende;

            // active l'utilisation de police embarquée
            legende.embedFonts = true;

            // crée un objet de formatage
            formatage = new TextFormat();

            // taille de la police
            formatage.size = 12;

            // instanciation de la police embarquée
            var police:MaPolice = new MaPolice();

            // affectation de la police au formatage
            formatage.font = police.fontName;

            // affectation du formatage au champ texte
            legende.setTextFormat ( formatage );

            // rend le champ texte non selectionnable
            legende.selectable = false;

            // stocke la couleur passée en paramètre
            couleur = pCouleur;

            // dessine le bouton
            fondBouton.graphics.beginFill ( couleur, 1 );
            fondBouton.graphics.drawRect ( 0, 0, 40, 110 );

            // activation du mode bouton
            buttonMode = true;

            // désactivation des objets enfants
            mouseChildren = false;

            // affectation de la vitesse contrôlée
            affecteVitesse ( pVitesse );

            // création de l'objet Tween
            interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
    1, vitesse, true );

            // écoute de l'événement MouseEvent.CLICK
            addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

        }
```

En testant le code précédent, chaque bouton possède désormais une légende intégrant les contours de police :

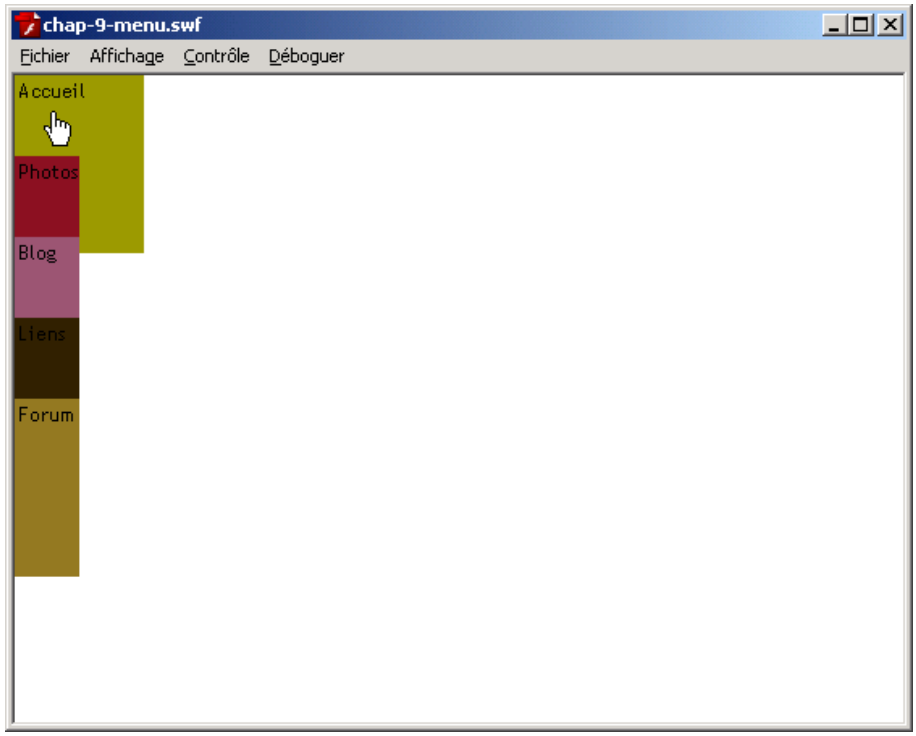

Figure 9-25. Champs texte formatés.

Il ne nous reste plus qu'à intégrer une gestion de la largeur et hauteur de chaque bouton. Pour cela nous ajoutons au constructeur de la classe `Bouton` deux paramètres `pLargeur` et `pHauteur` :

```
  public function Bouton ( pLargeur:Number, pHauteur:Number, pCouleur:Number,
pVitesse:Number, pLegende:String )

  {

        // ajoute chaque instance au tableau
        Bouton.tableauBoutons.push ( this );

        // création du fond du bouton
        fondBouton = new Shape();

        // ajout à la liste d'affichage
        addChild ( fondBouton );

        // crée le champ texte
        legende = new TextField();

        // redimensionnement automatique du champ texte
        legende.autoSize = TextFieldAutoSize.LEFT;

        // ajout à la liste d'affichage
        addChild ( legende );

        // affecte la légende
        legende.text = pLegende;

        // active l'utilisation de police embarquée
        legende.embedFonts = true;
```

```
            // crée un objet de formatage
            formatage = new TextFormat();

            // taille de la police
            formatage.size = 12;

            // instanciation de la police embarquée
            var police:MaPolice = new MaPolice();

            // affectation de la police au formatage
            formatage.font = police.fontName;

            // affectation du formatage au champ texte
            legende.setTextFormat ( formatage );

            // rend le champ texte non selectionnable
            legende.selectable = false;

            // stocke la couleur passée en paramètre
            couleur = pCouleur;

            // dessine le bouton
            fondBouton.graphics.beginFill ( couleur, 1 );
            fondBouton.graphics.drawRect ( 0, 0, pLargeur, pHauteur );

            // activation du mode bouton
            buttonMode = true;

            // désactivation des objets enfants
            mouseChildren = false;

            // affectation de la vitesse contrôlée
            affecteVitesse ( pVitesse );

            // création de l'objet Tween
            interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
    1, vitesse, true );

            // écoute de l'événement MouseEvent.CLICK
            addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

        }
```

Puis nous passons les valeurs voulues lors de la création de chaque bouton :

```
        // création des boutons
        monBouton = new Bouton( 60, 120, couleur, vitesse, legende );
```

La figure 9-26 illustre le résultat final :

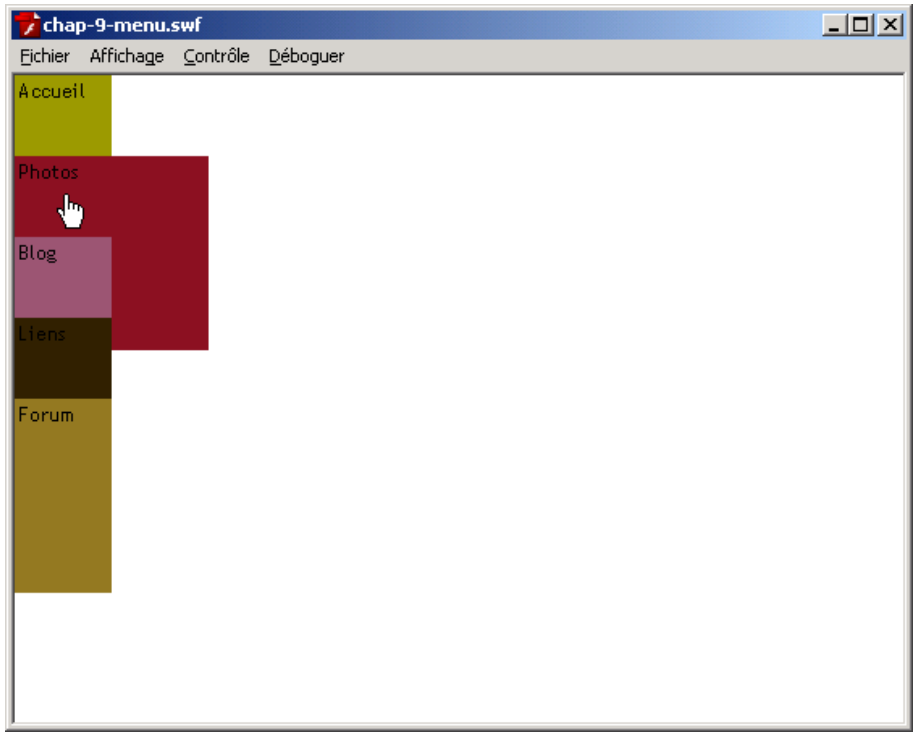

Figure 9-26. Boutons aux dimensions dynamiques.

Nous obtenons un menu simple à mettre en place et facilement modifiable. Nous pourrions rajouter d'autres fonctionnalités comme une adaptation automatique de la largeur du bouton par rapport à la légende. En réalité, nous pourrions ne jamais nous arrêter !

Dans beaucoup de projets, les boutons d'un menu sont généralement liés à des SWF. Lorsque le bouton est cliqué, le SWF correspondant est chargé, cela permet un meilleur découpage du site et un chargement à la demande. Nous allons modifier la classe `Bouton` afin de pouvoir stocker dans chaque bouton le nom du SWF correspondant. Nous ajoutons une propriété `swf` :

```
// swf associé
private var swf:String;
```

Puis nous modifions le constructeur afin d'accueillir le nom du SWF associé :

```
public function Bouton ( pLargeur:Number, pHauteur:Number, pSWF:String,
pCouleur:Number, pVitesse:Number, pLegende:String )

 {

      // ajoute chaque instance au tableau
      Bouton.tableauBoutons.push ( this );

      // création du fond du bouton
      fondBouton = new Shape();
```

```
            // ajout à la liste d'affichage
            addChild ( fondBouton );

            // crée le champ texte
            legende = new TextField();

            // redimensionnement automatique du champ texte
            legende.autoSize = TextFieldAutoSize.LEFT;

            // ajout à la liste d'affichage
            addChild ( legende );

            // affecte la légende
            legende.text = pLegende;

            // active l'utilisation de police embarquée
            legende.embedFonts = true;

            // crée un objet de formatage
            formatage = new TextFormat();

            // taille de la police
            formatage.size = 12;

            // instanciation de la police embarquée
            var police:MaPolice = new MaPolice();

            // affectation de la police au formatage
            formatage.font = police.fontName;

            // affectation du formatage au champ texte
            legende.setTextFormat ( formatage );

            // rend le champ texte non selectionnable
            legende.selectable = false;

            // stocke la couleur passée en paramètre
            couleur = pCouleur;

            // stocke le nom du SWF associé
            swf = pSWF;

            // dessine le bouton
            fondBouton.graphics.beginFill ( couleur, 1 );
            fondBouton.graphics.drawRect ( 0, 0, pLargeur, pHauteur );

            // activation du mode bouton
            buttonMode = true;

            // désactivation des objets enfants
            mouseChildren = false;

            // affectation de la vitesse contrôlée
            affecteVitesse ( pVitesse );

            // création de l'objet Tween
            interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut, 1,
      1, vitesse, true );

            // écoute de l'événement MouseEvent.CLICK
            addEventListener ( MouseEvent.ROLL_OVER, survolSouris );
```

```
}
```

Puis nous instancions les boutons du menu :

```
// import de la classe Bouton
import org.bytearray.ui.Bouton;

// création d'un conteneur pour le menu
var conteneurMenu:Sprite = new Sprite();

// tableau associatif contenant les données
var donnees:Array = new Array();

donnees.push ( { legende : "Accueil", vitesse : 1, swf : "accueil.swf",
couleur : 0x999900 } );
donnees.push ( { legende : "Photos", vitesse : 1, swf : "photos.swf", couleur
: 0x881122 } );
donnees.push ( { legende : "Blog", vitesse : 1, swf : "blog.swf", couleur :
0x995471 } );
donnees.push ( { legende : "Liens", vitesse : 1, swf : "liens.swf", couleur :
0xCC21FF } );
donnees.push ( { legende : "Forum", vitesse : 1, swf : "forum.swf", couleur :
0x977821 } );

// nombre de rubriques
var lng:int = donnees.length;

var monBouton:Bouton;
var legende:String;
var couleur:Number;
var vitesse:Number;
var swf:String;

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

{

  // récupération des infos
  legende = donnees[i].legende;
  couleur = donnees[i].couleur;
  vitesse = donnees[i].vitesse;
  swf = donnees[i].swf;

  // création des boutons
  monBouton = new Bouton( 60, 120, swf, couleur, vitesse, legende );

  // positionnement
  monBouton.y = 50 * i;

  // ajout au sein du conteneur
  conteneurMenu.addChild ( monBouton );

}

// affichage du menu
addChild ( conteneurMenu );
```

Chaque bouton est ainsi lié à un SWF. Lors du chapitre 7 nous avons crée un menu dynamique qui nous redirigeait à une adresse spécifique en ouvrant une nouvelle fenêtre navigateur. Pour cela nous stockions

le lien au sein d'une propriété du bouton, puis au moment du clic nous accédions à celle-ci.

Voici le code final de la classe `Bouton` :

```
package org.bytearray.ui

{

  import flash.display.Shape;
  import flash.display.Sprite;
  import flash.text.Font;
  import flash.text.TextFormat;
  // import des classes liées Tween au mouvement
  import fl.transitions.Tween;
  import fl.transitions.easing.Bounce;
  // import de la classe MouseEvent
  import flash.events.MouseEvent;
  // import de la classe TextField et TextFieldAutoSize
  import flash.text.TextField;
  import flash.text.TextFieldAutoSize;

  public class Bouton extends Sprite

  {

        // stocke le fond du bouton
        private var fondBouton:Shape;
        // stocke l'objet Tween pour les différents état du bouton
        private var interpolation:Tween;
        // stocke les références aux boutons
        private static var tableauBoutons:Array = new Array();
        // stocke la couleur en cours du bouton
        private var couleur:Number;
        // stocke la vitesse d'ouverture de chaque bouton
        private var vitesse:Number;
        // légende du bouton
        private var legende:TextField;
        // formatage des légendes
        private var formatage:TextFormat;
        // swf associé
        private var swf:String;

        public function Bouton ( pLargeur:Number, pHauteur:Number,
pSWF:String, pCouleur:Number, pVitesse:Number, pLegende:String )

        {

              // ajoute chaque instance au tableau
              Bouton.tableauBoutons.push ( this );

              // création du fond du bouton
              fondBouton = new Shape();

              // ajout à la liste d'affichage
              addChild ( fondBouton );

              // crée le champ texte
              legende = new TextField();

              // redimensionnement automatique du champ texte
```

```
            legende.autoSize = TextFieldAutoSize.LEFT;

            // ajout à la liste d'affichage
            addChild ( legende );

            // affecte la légende
            legende.text = pLegende;

            // active l'utilisation de police embarquée
            legende.embedFonts = true;

            // crée un objet de formatage
            formatage = new TextFormat();

            // taille de la police
            formatage.size = 12;

            // instanciation de la police embarquée
            var police:MaPolice = new MaPolice();

            // affectation de la police au formatage
            formatage.font = police.fontName;

            // affectation du formatage au champ texte
            legende.setTextFormat ( formatage );

            // rend le champ texte non selectionnable
            legende.selectable = false;

            // stocke la couleur passée en paramètre
            couleur = pCouleur;

            // stocke le nom du SWF
            swf = pSWF;

            // dessine le bouton
            fondBouton.graphics.beginFill ( couleur, 1 );
            fondBouton.graphics.drawRect ( 0, 0, pLargeur, pHauteur );

            // activation du mode bouton
            buttonMode = true;

            // désactivation des objets enfants
            mouseChildren = false;

            // affectation de la vitesse contrôlée
            affecteVitesse ( pVitesse );

            // création de l'objet Tween
            interpolation = new Tween ( fondBouton, "scaleX", Bounce.easeOut,
1, 1, vitesse, true );

            // écoute de l'événement MouseEvent.CLICK
            addEventListener ( MouseEvent.ROLL_OVER, survolSouris );

      }

      // déclenché lors du survol du bouton
      private function survolSouris ( pEvt:MouseEvent ):void

      {
            // stocke la longueur du tableau
```

```
                var lng:int = Bouton.tableauBoutons.length;

                for (var i:int = 0; i<lng; i++ )
Bouton.tableauBoutons[i].fermer();

                // démarrage de l'animation
                interpolation.continueTo ( 2, vitesse );

        }

        // méthode permettant de refermer le bouton
        private function fermer  ():void

        {
                // referme le bouton
                interpolation.continueTo ( 1, vitesse );

        }

        // gère l'affectation de la vitesse
        public function affecteVitesse ( pVitesse:Number ):void

        {

                // affecte la vitesse
                if ( pVitesse >= 1 && pVitesse <= 10 ) vitesse = pVitesse;

                else

                {
                        trace("Erreur : Vitesse non correcte, la valeur doit être
comprise entre 1 et 10");
                        vitesse = 1;

                }

        }

  }

}
```

Cette approche fonctionne sans problème et convient dans beaucoup de situations, en revanche lors de l'utilisation de classes personnalisées comme dans cette application nous allons externaliser les informations nécessaires grâce au modèle événementiel. De cette manière les objets « parleront » entre eux de manière faiblement couplée.

Nous allons épouser le même modèle que les objets natifs d'ActionScript 3, chaque bouton pourra diffuser un événement spécifique nous renseignant sur sa couleur, sa vitesse ainsi que le SWF correspondant.

Certains d'entre vous ont peut être déjà devinés vers quelle nouvelle notion nous nous dirigeons, en route pour le chapitre suivant intitulé *Diffusion d'événements personnalisés*.