13

Chargement du contenu

LA CLASSE LOADER **2**
CHARGER UN ÉLÉMENT EXTERNE 2
LA CLASSE LOADERINFO **5**
INTERAGIR AVEC LE CONTENU **18**
CRÉER UNE GALERIE **26**
LA COMPOSITION 29
REDIMENSIONNEMENT AUTOMATIQUE 34
GESTION DU LISSAGE 44
PRÉCHARGER LE CONTENU **48**
INTERROMPRE LE CHARGEMENT **57**
COMMUNIQUER ENTRE DEUX ANIMATIONS **58**
MODÈLE DE SÉCURITÉ DU LECTEUR FLASH **61**
PROGRAMMATION CROISÉE 62
UTILISER UN FICHIER DE RÉGULATION 63
CONTEXTE DE CHARGEMENT **65**
CONTOURNER LES RESTRICTIONS DE SÉCURITÉ **66**
BIBLIOTHÈQUE PARTAGÉE **72**
DÉSACTIVER UNE ANIMATION CHARGÉE **79**
COMMUNICATION AVM1 ET AVM2 ..... **81**

La classe Loader

ActionScript 3 élimine les nombreuses fonctions et méthodes existantes dans les précédentes versions d'ActionScript telles `loadMovie` ou `loadMovieNum` et étend le concept de la classe `MovieClipLoader` introduite avec Flash MX 2004.

Thibault Imbert
pratiqueactionscript3.bytearray.org

Afin de charger différents types de contenus tels des images ou des animations nous utilisons la classe `flash.display.Loader`. Celle-ci hérite de la classe `DisplayObjectContainer` et peut donc être ajoutée à la liste d'affichage.

D'autres classes telles `flash.net.URLLoader` ou `flash.net.URLStream` peuvent être utilisées afin de charger tout type de contenu, mais leur utilisation s'avère plus complexe. Nous reviendrons en détail sur ces classes au cours du chapitre 14 intitulé *Charger et envoyer des données*.

Nous allons commencer notre apprentissage en chargeant différents types de contenu, puis nous continuerons notre exploration de la classe `Loader` en créant une galerie photo.

A retenir

- La classe `Loader` permet le chargement de contenu graphique seulement.
- Celle-ci hérite de la classe `DisplayObjectContainer` et peut donc être ajoutée à la liste d'affichage.
- Il faut considérer la classe `Loader` comme un objet d'affichage.

Charger un élément externe

La classe `Loader` permet de charger des images au format PNG, JPEG, JPEG progressif et GIF non animé. Des animations SWF peuvent aussi être chargées, nous verrons comment communiquer avec celles-ci en fin de chapitre.

Afin de charger un de ces fichiers, nous instancions tout d'abord la classe `Loader` :

```
var chargeur:Loader = new Loader();
```

Puis nous appelons la méthode `load` dont voici la signature :

```
public function load(request:URLRequest, context:LoaderContext = null):void
```

Celle-ci requiert deux paramètres dont voici le détail :

- `request` : un objet `URLRequest` contenant l'adresse de l'élément à charger.
- `context` : un objet `LoaderContext` définissant le contexte de l'élément chargé.

La classe `URLRequest` est utilisée pour toute connexion externe HTTP liée aux classes `URLLoader`, `URLStream`, `Loader` et `FileReference`.

Dans le code suivant, nous enveloppons l'URL à atteindre dans un objet `URLRequest` :

```
// création du chargeur
var chargeur:Loader = new Loader();

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load ( maRequete );
```

L'utilisation d'un objet `URLRequest` diffère des précédentes versions d'ActionScript où une simple chaîne de caractères était passée aux différentes fonctions liées au chargement externe. Certaines classes ActionScript 3 telles `Socket`, `URLStream`, `NetConnection` et `NetStream` font néanmoins exception et ne nécessitent pas d'objet `URLRequest`.

Nous reviendrons en détail sur la classe `URLRequest` au cours du chapitre 14 intitulé *Charger et envoyer des données*.

Nous ne nous intéressons pas pour le moment au paramètre `context` qui n'est pas utilisé par défaut. Nous verrons plus loin dans ce chapitre l'intérêt de ce dernier.

Afin de rendre graphiquement le contenu chargé nous devons ajouter l'objet `Loader` à la liste d'affichage :

```
// création du chargeur
var chargeur:Loader = new Loader();

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

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

L'image est alors affichée :

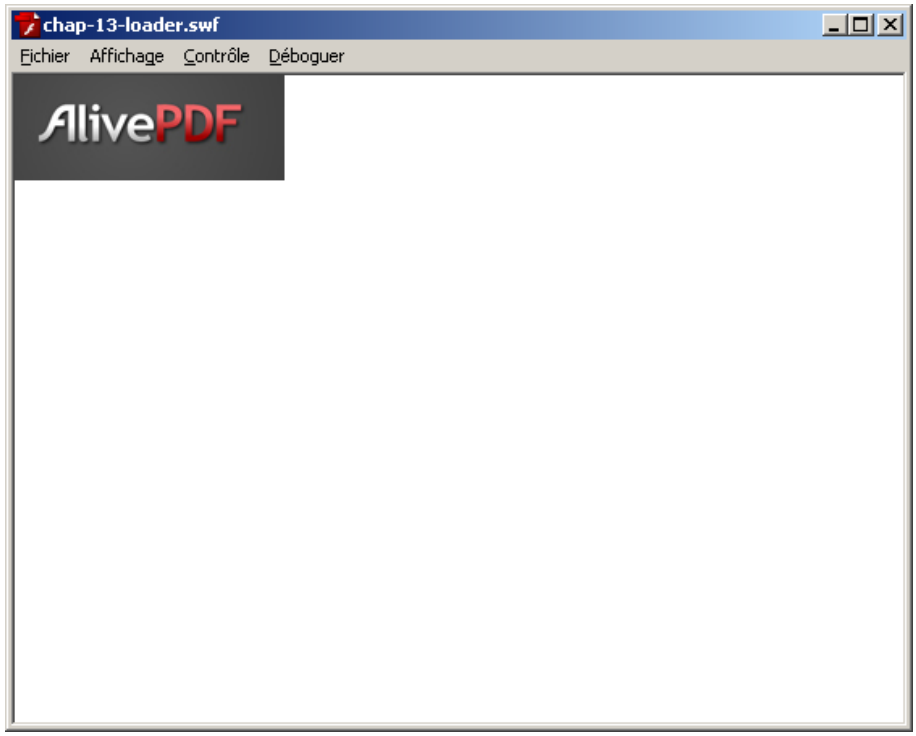

Figure 13-1. Image chargée dynamiquement.

Il est important de signaler qu'une instance de la classe `Loader` ne peut contenir qu'un seul enfant. Ainsi, lorsque la méthode `load` est exécutée, le contenu précédent est automatiquement remplacé.

Ainsi, les images bitmap sont automatiquement supprimées de la mémoire, lorsque celles-ci sont déchargées. Ce n'est pas le cas des animations dont nous devons gérer la désactivation complète. Nous reviendrons sur cette contrainte en fin de chapitre.

Comme depuis toujours, le chargement des données externes en ActionScript est asynchrone, cela signifie que l'instruction `load` n'est pas bloquante. Le code continue d'être exécuté une fois le chargement démarré. Nous serons avertis de la fin du chargement des données plus tard dans le temps grâce à un événement approprié.

A retenir

- La méthode `load` permet de charger le contenu externe.
- Afin de voir le contenu chargé, l'objet `Loader` doit être ajouté à la liste d'affichage.

La classe LoaderInfo

Afin de pouvoir accéder au contenu chargé, ou d'être averti des différentes phases du chargement nous utilisons les événements diffusés par la classe `flash.display.LoaderInfo`. Celle-ci contient des informations relatives au contenu chargé de type bitmap ou SWF.

Il existe deux moyens d'accéder à un objet `LoaderInfo` :

- Par la propriété `contentLoaderInfo` de l'objet `Loader`.
- Par la propriété `loaderInfo` de n'importe quel `DisplayObject`.

Lorsque nous accédons à la propriété `contentLoaderInfo` de la classe `Loader` nous récupérons une référence à l'objet `LoaderInfo` gérant le chargement de l'élément.

Si nous tentons d'écouter les différents événements liés au chargement directement auprès de l'objet `Loader`, aucun événement n'est diffusé :

```
// création du chargeur
var chargeur:Loader = new Loader();

// tentative d'écoute de l'événement Event.COMPLETE
chargeur.addEventListener ( Event.COMPLETE, contenuCharge );

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

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

function contenuCharge ( pEvt:Event ):void

{

  trace( pEvt );

}
```

Ainsi, l'écoute des différents événements propres au chargement, se fait auprès de l'objet `LoaderInfo` accessible par la propriété `contentLoaderInfo` de l'objet `Loader` :

```
// création du chargeur
```

```
var chargeur:Loader = new Loader();

// écoute de l'événement Event.COMPLETE
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, contenuCharge
);

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

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

function contenuCharge ( pEvt:Event ):void

{

  // affiche : [Event type="complete" bubbles=false cancelable=false
eventPhase=2]
  trace( pEvt );

  // affiche : [object LoaderInfo]
  trace( pEvt.target );

}
```

De nombreux événements sont diffusés par la classe `LoaderInfo`, voici le détail de chacun d'entre eux :

- `Event.OPEN` : diffusé lorsque le lecteur commence à charger le contenu.
- `ProgressEvent.PROGRESS` : diffusé lorsque le chargement est en cours. Celui-ci renseigne sur le nombre d'octets chargés et totaux. Sa fréquence de diffusion est liée à la vitesse de téléchargement de l'élément.
- `Event.INIT` : diffusé lorsqu'il est possible d'accéder au code d'un SWF chargé. Cet événement n'est pas utilisé dans le cas de chargement d'image bitmap.
- `Event.COMPLETE` : diffusé lorsque le chargement est terminé.
- `Event.UNLOAD` : diffusé lorsque la méthode `unload` est appelée ou qu'un nouvel élément va être chargé pour remplacer le précédent.
- `IOErrorEvent.IO_ERROR` : diffusé lorsque le chargement échoue.
- `HTTPStatusEvent.HTTP_STATUS` : indique le code d'état de la requête HTTP.

Afin de gérer avec finesse le chargement nous pouvons écouter chacun des événements :

```
// création du chargeur
var chargeur:Loader = new Loader();

// référence à l'objet LoaderInfo
var cli:LoaderInfo = chargeur.contentLoaderInfo;
```

```
// écoute des événements liés au chargement
cli.addEventListener ( Event.OPEN, debutChargement );
cli.addEventListener ( Event.INIT, initialisation );
cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
cli.addEventListener ( Event.COMPLETE, chargementTermine );
cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );
cli.addEventListener ( HTTPStatusEvent.HTTP_STATUS, echecHTTP );
cli.addEventListener ( Event.UNLOAD, suppressionContenu );

// url à atteindre
var maRequete:URLRequest = new URLRequest ("photo.jpg");

// chargement du contenu
chargeur.load( maRequete );

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

function debutChargement ( pEvt:Event ):void

{

  // affiche : [Event type="open" bubbles=false cancelable=false
eventPhase=2]
  trace( pEvt );

}

function initialisation ( pEvt:Event ):void

{

  // affiche : [Event type="init" bubbles=false cancelable=false
eventPhase=2]
  trace( pEvt );

}

function chargement ( pEvt:ProgressEvent ):void

{

  // affiche : [ProgressEvent type="progress" bubbles=false cancelable=false
eventPhase=2 bytesLoaded=0 bytesTotal=5696]
  trace( pEvt );

}

function chargementTermine ( pEvt:Event ):void

{

  // affiche : [Event type="complete" bubbles=false cancelable=false
eventPhase=2]
  trace( pEvt );

}

function echecChargement ( pEvt:IOErrorEvent ):void

{
```

```
  trace( pEvt );

}

function echecHTTP ( pEvt:HTTPStatusEvent ):void

{

  trace( pEvt );

}

function suppressionContenu ( pEvt:Event ):void

{

  // affiche : [Event type="unload" bubbles=false cancelable=false
eventPhase=2]
  trace( pEvt );

}
```

Si nous tentons d'accéder au contenu chargé avant l'événement `Event.COMPLETE` l'accès à la propriété `content` lève une erreur à l'exécution. Dans le code suivant nous tentons d'accéder au contenu durant l'événement `ProgressEvent.PROGRESS` :

```
function chargement ( pEvt:ProgressEvent ):void

{

  trace( pEvt.target.content );

}
```

L'erreur suivante est levée à l'exécution :

```
Error: Error #2099: L'objet en cours de chargement n'est pas suffisamment
chargé pour fournir ces informations.
```

Une fois le contenu totalement chargé, nous y accédons grâce à la propriété `content` de la classe `LoaderInfo` :

```
function chargementTermine ( pEvt:Event ):void

{

  // affiche : [object Bitmap]
  trace( pEvt.target.content );

}
```

Mais aussi par la propriété `content` de l'objet `Loader` :

```
// affiche : [object Bitmap]
trace( chargeur.content ) ;
```

A l'aide des différentes propriétés de la classe `LoaderInfo` nous obtenons différents types d'informations :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // accès aux informations liées au contenu chargé
  var contenu:Bitmap = Bitmap ( objetLoaderInfo.content );

  var largeur:Number = objetLoaderInfo.width;
  var hauteur:Number = objetLoaderInfo.height;

  var urlContenu:String = objetLoaderInfo.url;

  var type:String = objetLoaderInfo.contentType;

  // affiche : [object Bitmap]
  trace ( contenu );

  // affiche : 167 65
  trace ( largeur, hauteur );

  // affiche :
file:///K|/Work/Bytearray.org/O%27Reilly/Pratique%20d%27ActionScript/Chapitre
%2013/flas/photo.jpg
  trace ( urlContenu );

  // affiche : image/jpeg
  trace ( type );

}
```

Lors du chapitre 12 intitulé *Programmation Bitmap*, nous avons vu que toutes les données bitmap étaient assimilées à des objets `BitmapData`. Dans le cas d'une image chargée dynamiquement, les données bitmap sont enveloppées au préalable par un objet `flash.display.Bitmap` afin de pouvoir être rendues.

Dans le code suivant, nous récupérons les dimensions de l'image chargée ainsi que les données bitmap la composant :

```
function chargementTermine ( pEvt:Event ):void

{

  // référence le contenu
  var contenu:DisplayObject = pEvt.target.content;

  // si le contenu chargé est une image
  if ( contenu is Bitmap )

  {

       // transtype en tant qu'image
       var image:Bitmap = Bitmap ( contenu );

       // affiche : 167 65
       trace( image.width, image.height );

       //affiche : [object BitmapData]
```

```
        trace( image.bitmapData );

    }

}
```

Nous pouvons effacer une partie de l'image chargée, en peignant directement dessus à l'aide de la méthode `fillRect` de la classe `BitmapData` :

```
function chargementTermine ( pEvt:Event ):void

{

    // référence le contenu
    var contenu:DisplayObject = pEvt.target.content;

    // si le contenu chargé est une image
    if ( contenu is Bitmap )

    {

        // transtype en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        var donneesBitmap:BitmapData = image.bitmapData;

        donneesBitmap.fillRect( new Rectangle ( 15, 15, 135, 30 ), 0xC8BCA4 );

    }

}
```

La figure 13-2 illustre le résultat :

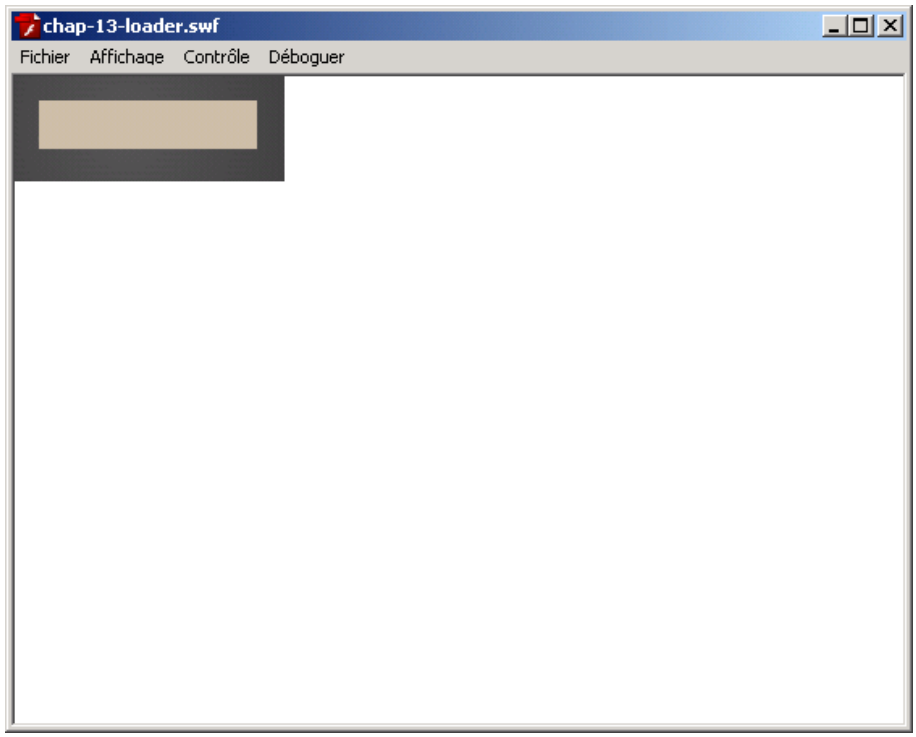

Figure 13-2. Modification des données bitmap.

La partie centrale de l'image chargée est peinte de pixels beiges.

Souvenez-vous que les événements sont diffusés depuis l'objet `LoaderInfo`. Ainsi la propriété `target` ne référence pas l'objet `Loader` mais l'objet `LoaderInfo` associé à l'élément chargé.

Afin d'accéder à l'objet `Loader` associé à l'objet `LoaderInfo`, nous utilisons sa propriété `loader` :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // affiche : true
  trace( objetLoaderInfo.loader == chargeurImage );

}
```

De la même manière, la propriété `contentLoaderInfo` de l'objet `Loader` pointe vers le même objet `LoaderInfo` que celui référencé par la propriété `loaderInfo` de l'élément chargé :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  var objetLoader:Loader = pEvt.target.loader;

  // affiche : true
  trace( objetLoader.contentLoaderInfo == objetLoaderInfo.content.loaderInfo
);

}
```

L'objet `LoaderInfo` est donc partagé entre deux environnements :

L'animation chargeant le contenu et le contenu lui-même.

Il est important de noter que bien qu'il s'agisse d'une sous-classe de `DisplayObjectContainer`, la classe `Loader` ne possède pas toutes les capacités propres à ce type.

Certaines propriétés telles `numChildren` peuvent être utilisées :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoader:Loader = pEvt.target.loader;

  // affiche : true
```

```
  trace( objetLoader.numChildren );

}
```

Il est en revanche impossible d'appeler les différentes méthodes de manipulation d'objets enfants sur celle-ci. Dans le code suivant nous tentons d'appeler la méthode `removeChildAt` sur l'objet `chargeurImage` :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoader:Loader = pEvt.target.loader;

  // tentative de suppression du premier enfant de l'objet Loader
  objetLoader.removeChildAt(0);

}
```

L'erreur à l'exécution suivante est levée :

```
// affiche : Error: Error #2069: La classe Loader ne met pas en oeuvre cette
méthode.
```

Afin de supprimer l'image chargée, nous devons appeler la méthode `unload` de la classe `Loader`.

Dans le code suivant, nous supprimons l'image chargée aussitôt le chargement terminé :

```
function chargementTermine ( pEvt:Event ):void

{

  // référence le contenu
  var contenu:DisplayObject = pEvt.target.content;

  // si le contenu chargé est une image
  if ( contenu is Bitmap )

  {

        // transtype en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        var donneesBitmap:BitmapData = image.bitmapData;

        donneesBitmap.fillRect( new Rectangle ( 15, 15, 135, 30 ), 0x009900 );

        pEvt.target.loader.unload();

  }

}
```

Aussitôt le contenu supprimé, l'événement `Event.UNLOAD` est diffusé.

En cas d'erreur de chargement, nous devons gérer et indiquer à l'utilisateur qu'une erreur s'est produite. Pour cela nous devons obligatoirement écouter l'événement `IOErrorEvent.IO_ERROR`.

Ce dernier est diffusé lorsque le contenu n'a pu être chargé, pour des raisons d'accès ou de compatibilité. Si nous tentons de charger un contenu non géré et que l'événement `IOErrorEvent.IO_ERROR` n'est pas écouté, une erreur à l'exécution est levée.

> Attention, si le bouton retour du navigateur est cliqué pendant le chargement de contenu externe par le lecteur Flash. L'événement `IOErrorEvent.IO_ERROR` est diffusé, alors que le contenu est pourtant compatible ou disponible.
>
> Il convient donc de *toujours* écouter cet événement.

Dans le code suivant, un fichier non compatible est chargé, l'événement `IOErrorEvent.IO_ERROR` est diffusé :

```
// création du chargeur
var chargeur:Loader = new Loader();

// référence à l'objet LoaderInfo
var cli:LoaderInfo = chargeur.contentLoaderInfo;

// écoute des événements liés au chargement
cli.addEventListener ( Event.OPEN, debutChargement );
cli.addEventListener ( Event.INIT, initialisation );
cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
cli.addEventListener ( Event.COMPLETE, chargementTermine );
cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );
cli.addEventListener ( HTTPStatusEvent.HTTP_STATUS, echecHTTP );
cli.addEventListener ( Event.UNLOAD, suppressionContenu );

// tentative de chargement d'un document PDF
// entraîne la diffusion de l'événement IOErrorEvent.IO_ERROR
var maRequete:URLRequest = new URLRequest ("document.pdf");

// chargement du contenu
chargeur.load( maRequete );

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

function debutChargement ( pEvt:Event ):void

{

  trace( pEvt );

}

function initialisation ( pEvt:Event ):void

{
```

```
  trace( pEvt );
}
function chargement ( pEvt:ProgressEvent ):void
{
  trace( pEvt );
}
function chargementTermine ( pEvt:Event ):void
{
  trace( pEvt );
}
function suppressionContenu ( pEvt:Event ):void
{
  trace( pEvt );
}
function echecChargement ( pEvt:Event ):void
{
  // affiche : [IOErrorEvent type="ioError" bubbles=false cancelable=false
eventPhase=2 text="Error #2124: Le type du fichier chargé est inconnu. URL:
file:///D|/Work/Bytearray.org/O%27Reilly/Pratique%20d%27ActionScript/Chapitre
%2013/flas/document.pdf"]
  trace ( pEvt );
}
```

Lorsqu'une animation est chargée par un objet `Loader`, l'objet `LoaderInfo` active de nouvelles propriétés liées au SWF.

Voici le détail de chacune d'entre elles :

- `LoaderInfo.frameRate` : Indique la cadence du SWF chargé.
- `LoaderInfo.`applicationDomain : retourne une référence à l'objet `ApplicationDomain` contenant les définitions de classe du SWF chargé.
- `LoaderInfo.`actionScriptVersion : indique la version d'ActionScript du SWF chargé.
- `LoaderInfo.parameters` : objet contenant les FlashVars associés au SWF chargé.
- `LoaderInfo.swfVersion` : indique la version du SWF chargé.

Toute tentative d'accès, à l'une de ces propriétés lorsqu'une image est chargée, se traduit par un échec. Dans le code suivant, une image est chargée, nous tentons alors d'accéder à la propriété `frameRate` :

```
function chargementTermine ( pEvt:Event ):void

{

  // tentative d'accès à la cadence de l'élément chargé
  trace( pEvt.target.frameRate );

}
```

L'erreur à l'exécution suivante est levée :

```
Error: Error #2098: L'objet en cours de chargement n'est pas un fichier .swf,
vous ne pouvez donc pas en extraire des propriétés SWF.
```

Afin d'utiliser ces propriétés nous chargeons une animation :

```
// url à atteindre
var maRequete:URLRequest = new URLRequest ("animation.swf");
```

Puis nous accédons au sein de la fonction `chargementTermine` aux différentes informations liées au SWF :

```
function chargementTermine ( pEvt:Event ):void

{


  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // affiche : 30
  trace( objetLoaderInfo.frameRate );

  // affiche : [object ApplicationDomain]
  trace( objetLoaderInfo.applicationDomain  );

  // affiche : 3
  trace( objetLoaderInfo.actionScriptVersion );

  // affiche : [object Object]
  trace( objetLoaderInfo.parameters );

  // affiche : 9
  trace( objetLoaderInfo.swfVersion );

}
```

Attention, il convient de toujours utiliser les propriétés `width` et `height` de l'objet `LoaderInfo` afin de récupérer la largeur et hauteur du contenu chargé. Dans le cas de chargement d'animations, la propriété `content` référence le scénario principal du SWF.

Ainsi, l'utilisation des propriétés `width` et `height` renvoient la taille du contenu de l'animation, et non les dimensions du SWF :

```
function chargementTermine ( pEvt:Event ):void
```

```
{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  var contenu:DisplayObject = objetLoaderInfo.content;

  // accède aux dimensions du contenu de l'animation
  // affiche : 187.5 187.5
  trace( contenu.width, contenu.height );

  // accède aux dimensions du SWF
  // affiche : 550 400
  trace( objetLoaderInfo.width, objetLoaderInfo.height );

}
```

Nous pouvons tester lors de la fin du chargement le type de contenu chargé à l'aide du mot clé `is` afin de déterminer si le contenu chargé est une animation ou une image

En testant le type de la propriété `content` nous pouvons déterminer le type de contenu chargé :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  var contenu:DisplayObject = objetLoaderInfo.content;

  if ( contenu is Bitmap )

  {

       trace("Une image a été chargée !");

  } else

  {

       trace("Une animation a été chargée !");

  }

}
```

Nous venons de traiter l'utilisation de la classe `LoaderInfo` à travers la propriété `contentLoaderInfo` de l'objet `Loader`. A l'inverse lorsque nous accédons à la propriété `loaderInfo` d'un `DisplayObject` nous accédons aux informations du SWF contenant le `DisplayObject`.

A l'instar des propriétés `stage` et `root`, la propriété `loaderInfo` renvoie `null`, tant que l'objet graphique n'est pas présent au sein de la liste d'affichage :

```
// création d'un objet graphique
var monSprite:Sprite = new Sprite();

// affiche : null
trace( monSprite.loaderInfo );

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

// affiche : [object LoaderInfo]
trace( monSprite.loaderInfo );

// affiche : true
trace( loaderInfo == monSprite.loaderInfo );
```

Afin d'accéder aux différentes propriétés de la classe `LoaderInfo`. Nous devons attendre la fin du chargement du SWF en cours. Si nous tentons d'accéder aux propriétés de la classe `LoaderInfo` avant l'événement `Event.INIT` :

```
trace( monSprite.loaderInfo.width );
```

L'erreur suivante est levée à l'exécution :

```
Error: Error #2099: L'objet en cours de chargement n'est pas suffisamment
chargé pour fournir ces informations.
```

A l'aide de l'événement `Event.INIT` ou `Event.COMPLETE` nous pouvons accéder correctement aux informations du SWF en cours :

```
// création d'un objet graphique
var monSprite:Sprite = new Sprite();

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

// écoute de l'événement Event.COMPLETE auprès de
// l'objet LoaderInfo du SWF en cours
monSprite.loaderInfo.addEventListener ( Event.COMPLETE, chargementTermine );

function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // affiche : [object LoaderInfo]
  trace( objetLoaderInfo );

  // affiche : 550 400
  trace( objetLoaderInfo.width, objetLoaderInfo.height );

  // affiche : application/x-shockwave-flash
  trace( objetLoaderInfo.contentType );

  // affiche : 12
  trace( objetLoaderInfo.frameRate );

  // affiche : 577 577
  trace ( objetLoaderInfo.bytesLoaded, objetLoaderInfo.bytesTotal );
```

```
}
```

Nous pouvons aussi référencer l'objet `LoaderInfo` par la propriété `loaderInfo` du scénario principal :

```
// écoute de l'événement Event.COMPLETE auprès de
// l'objet LoaderInfo du SWF en cours
loaderInfo.addEventListener ( Event.COMPLETE, chargementTermine );
```

De cette manière, un SWF peut gérer son préchargement de manière autonome. Ainsi, chaque SWF possède un objet `LoaderInfo` associé regroupant les informations relatives à celui-ci.

Nous retrouvons avec le code précédent, l'équivalent de l'événement `onLoad` existant en ActionScript 1 et 2.

A retenir

- La classe `LoaderInfo` contient des informations relatives à un élément chargé de type bitmap ou SWF.
- La propriété `contentLoaderInfo` de l'objet `Loader` référence l'objet `LoaderInfo` associé à l'élément chargé.
- Tous les objets de type `DisplayObject` possèdent une propriété `loaderInfo` associé au SWF en cours.
- Lorsqu'une image est chargée, seules certaines propriétés de la classe `LoaderInfo` sont utilisables.
- Lorsqu'un SWF est chargé, la totalité des propriétés de la classe `LoaderInfo` sont utilisables.

Interagir avec le contenu

Il est généralement courant, de devoir interagir avec le contenu chargé. La classe `Loader` est un `DisplayObjectContainer` et hérite donc de la classe `InteractiveObject` facilitant l'interactivité souris et clavier. Il est cependant impossible d'activer la propriété `buttonMode` auprès de l'objet `Loader`, l'interactivité est donc réduite.

Une première technique consiste à déplacer le contenu de l'objet `Loader` à l'aide de la méthode `addChild`.

Dans le code suivant, nous accédons au contenu chargé puis nous le déplaçons au sein d'un autre conteneur :

```
function chargementTermine ( pEvt:Event ):void

{

  // référence le contenu
  var contenu:DisplayObject = pEvt.target.content;
```

```
        // affiche : [object Loader]
        trace( contenu.parent );

        // réattribution de conteneur, l'image quitte l'objet Loader
        addChild ( contenu );

        // affiche : [object MainTimeline]
        trace( contenu.parent );

    }
```

Nous référençons le contenu à l'aide de la propriété `content` de l'objet `LoaderInfo`. Il convient de rester le plus générique possible, nous utilisons donc le type commun `DisplayObject` propre à tout type de contenu chargé, puis nous déplaçons le contenu chargé à l'aide de la méthode `addChild`.

> Souvenez-vous, lorsqu'un `DisplayObject` déjà présent dans la liste d'affichage est passé à la méthode `addChild`, l'objet sur lequel la méthode est appelée devient le nouveau conteneur.
>
> C'est le concept de réattribution de parent couvert au sein du chapitre 4 intitulé *Liste d'affichage*.

Afin de dupliquer l'image chargée, nous n'avons pas à recharger l'image au sein d'un autre objet `Loader`. Nous copions les données bitmap puis les associons à un nouvel objet `Bitmap` :

```
    function chargementTermine ( pEvt:Event ):void

    {
        // référence le contenu
        var contenu:DisplayObject = pEvt.target.content;

        // si le contenu chargé est une image
        if ( contenu is Bitmap )

        {
            // transtype en tant qu'image
            var image:Bitmap = Bitmap ( contenu );

            var donneesBitmap:BitmapData = image.bitmapData;

            // création d'une copie des données bitmaps
            var copieDonneesBitmap:BitmapData = donneesBitmap.clone();

            var copieImage:Bitmap = new Bitmap ( copieDonneesBitmap );

            copieImage.x = image.width + 5;

            addChild ( image );
            addChild ( copieImage );
```

```
  }

}
```

La figure 13-3 illustre le résultat :

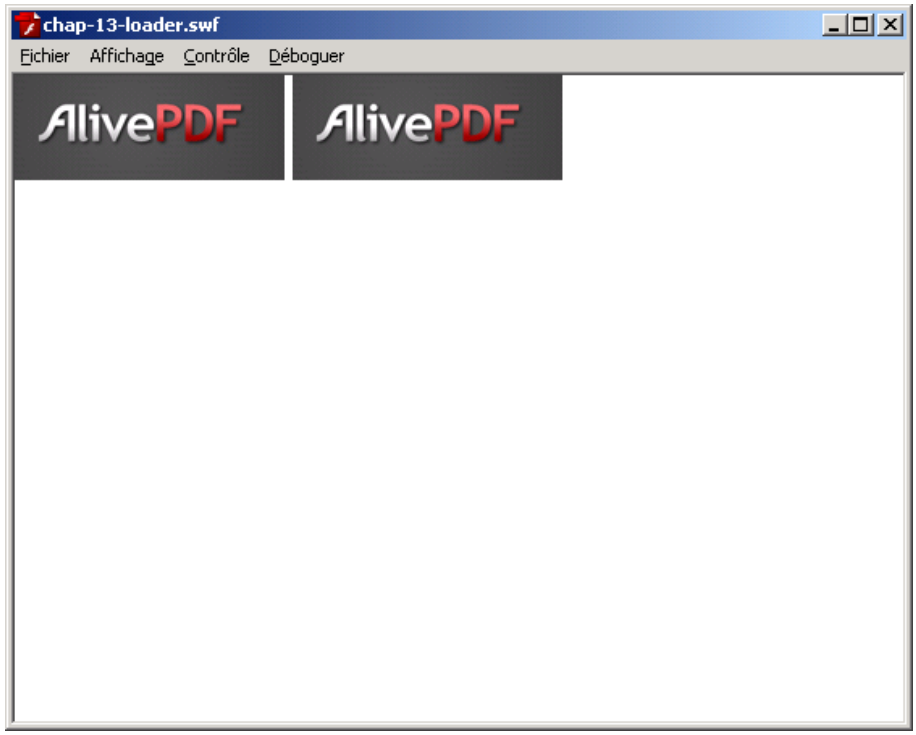

Figure 13-3. Copie de l'image chargée.

Afin de rendre les images cliquables, nous les ajoutons au sein d'objets interactifs tels `Sprite` :

```
function chargementTermine ( pEvt:Event ):void

{

  // référence le contenu
  var contenu:DisplayObject = pEvt.target.content;

  // si le contenu chargé est une image
  if ( contenu is Bitmap )

  {

        // transtype en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        var premierConteneur:Sprite = new Sprite();

        premierConteneur.buttonMode = true;

        premierConteneur.addChild( image );

        var donneesBitmap:BitmapData = image.bitmapData;
```

```
            // création d'une copie des données bitmaps
            var copieDonneesBitmap:BitmapData = donneesBitmap.clone();

            var copieImage:Bitmap = new Bitmap ( copieDonneesBitmap );

            var secondConteneur:Sprite = new Sprite();

            secondConteneur.buttonMode = true;

            secondConteneur.addChild( copieImage );

            secondConteneur.x = premierConteneur.width + 5;

            addChild ( premierConteneur );
            addChild ( secondConteneur );

            premierConteneur.addEventListener( MouseEvent.CLICK, clicImage );
            secondConteneur.addEventListener( MouseEvent.CLICK, clicImage );

        }

    }

    function clicImage ( pEvt:MouseEvent ):void

    {

      // affiche : [MouseEvent type="click" bubbles=true cancelable=false
    eventPhase=2 localX=48 localY=36 stageX=220 stageY=36 relatedObject=null
    ctrlKey=false altKey=false shiftKey=false delta=0]
      trace( pEvt );

    }
```

Les deux images bitmaps sont placées dans un conteneur `Sprite` spécifique, facilitant l'interactivité. Afin d'activer le comportement bouton, nous utilisons la propriété `buttonMode`.

Souvenez-vous, afin de rendre plus souple notre code, nous capturons l'événement `MouseEvent.CLICK` auprès du parent des événements réactifs. Nous préférons donc le code suivant :

```
    // création d'un conteneur pour les images
    var conteneurImages:Sprite = new Sprite();

    addChild ( conteneurImages );

    // capture de l'événement MouseEvent.MOUSE_CLICK auprès du conteneur
    conteneurImages.addEventListener( MouseEvent.CLICK, clicImages, true );

    function clicImages ( pEvt:MouseEvent ):void

    {

      // [MouseEvent type="click" bubbles=true cancelable=false eventPhase=1
    localX=87 localY=4 stageX=259 stageY=4 relatedObject=null ctrlKey=false
    altKey=false shiftKey=false delta=0]
      trace( pEvt );

    }
```

```
function chargementTermine ( pEvt:Event ):void

{

  // référence le contenu
  var contenu:DisplayObject = pEvt.target.content;

  // si le contenu chargé est une image
  if ( contenu is Bitmap )

  {

        // transtype en tant qu'image
        var image:Bitmap = Bitmap ( contenu );

        var premierConteneur:Sprite = new Sprite();

        premierConteneur.buttonMode = true;

        premierConteneur.addChild( image );

        var donneesBitmap:BitmapData = image.bitmapData;

        // création d'une copie des données bitmaps
        var copieDonneesBitmap:BitmapData = donneesBitmap.clone();

        var copieImage:Bitmap = new Bitmap ( copieDonneesBitmap );

        var secondConteneur:Sprite = new Sprite();

        secondConteneur.buttonMode = true;

        secondConteneur.addChild( copieImage );

        secondConteneur.x = premierConteneur.width + 5;

        conteneurImages.addChild ( premierConteneur );
        conteneurImages.addChild ( secondConteneur );

  }

}
```

Grâce à la capture de l'événement `MouseEvent.CLICK`, lorsque de nouvelles images seront placées ou supprimées du conteneur `conteneurImages`, aucune nouvelle souscription ou désinscription d'écouteurs ne sera nécessaire.

A chaque clic sur les images nous la supprimons de la liste d'affichage, pour cela nous modifions la fonction `clicImages` :

```
function clicImages ( pEvt:MouseEvent ):void

{

  pEvt.currentTarget.removeChild ( DisplayObject ( pEvt.target ) );

}
```

Lorsque l'image est cliquée, celle-ci est supprimée de la liste d'affichage. Une référence vers l'image d'origine demeure cependant au sein de l'objet `Loader`, il nous est donc impossible de libérer correctement les ressources.

De plus, en déplaçant le contenu de l'objet `Loader` nous modifions son fonctionnement interne. Ainsi, lors du prochain appel de la méthode `load`, celui-ci tentera de supprimer son contenu afin d'accueillir le nouveau. Celui-ci ayant été déplacé, l'objet `Loader` tente alors de supprimer un objet qui n'est plus son enfant, ce qui provoque alors une erreur à l'exécution.

> Ce comportement est un bug du lecteur Flash 9.0.115 qui sera corrigé au sein du prochain lecteur Flash 10.

Il est donc fortement déconseillé de déplacer le contenu de l'objet `Loader` sous peine de perturber le mécanisme interne de celui-ci. De manière générale nous préférons créer un objet `Loader` pour chaque contenu chargé.

Nous pouvons donc établir le bilan suivant :

```
// bonne pratique
addChild ( chargeur );

// mauvaise pratique
addChild ( chargeur.content );
```

Dans le code suivant, nous chargeons plusieurs images au sein de différents objets `Loader` :

```
var conteneurImages:Sprite = new Sprite();

addChild ( conteneurImages );

// capture de l'événement MouseEvent.MOUSE_CLICK auprès du conteneur
conteneurImages.addEventListener( MouseEvent.CLICK, clicImages, true );

var chargeurImage:Loader;

var requete:URLRequest = new URLRequest();

var tableauImages:Array = new Array ("photo1.jpg", "photo2.jpg",
"photo3.jpg", "photo4.jpg", "photo5.jpg", "photo3.jpg");

var lng:int = tableauImages.length;

var colonnes:int = 4;

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

{

  // création du chargeur
  chargeurImage = new Loader();
```

```
  requete.url = tableauImages[i];

  chargeurImage.load ( requete );

  chargeurImage.x = Math.round (i % colonnes) * 120;
  chargeurImage.y = Math.floor (i / colonnes) * 80

  conteneurImages.addChild ( chargeurImage );

}

function clicImages ( pEvt:MouseEvent ):void

{

  // [MouseEvent type="click" bubbles=true cancelable=false eventPhase=1
localX=10 localY=49 stageX=250 stageY=49 relatedObject=null ctrlKey=false
altKey=false shiftKey=false delta=0]
  trace( pEvt );

}
```

Afin de rendre les images cliquables, nous les imbriquons dans des conteneurs de type `Sprite` :

```
var conteneurImages:Sprite = new Sprite();

addChild ( conteneurImages );

// capture de l'événement MouseEvent.MOUSE_CLICK auprès du conteneur
conteneurImages.addEventListener( MouseEvent.CLICK, clicImages, true );

var chargeurImage:Loader;

var conteneurLoader:Sprite;

var requete:URLRequest = new URLRequest();

var tableauImages:Array = new Array ("photo1.jpg", "photo2.jpg",
"photo3.jpg", "photo4.jpg", "photo5.jpg", "photo3.jpg");

var lng:int = tableauImages.length;

var colonnes:int = 4;

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

{

  // création du chargeur
  chargeurImage = new Loader();

  conteneurLoader = new Sprite();

  conteneurLoader.addChild( chargeurImage );

  requete.url = tableauImages[i];

  chargeurImage.load ( requete );

  conteneurLoader.x = Math.round (i % colonnes) * 120;
```

```
        conteneurLoader.y = Math.floor (i / colonnes) * 80

        conteneurImages.addChild ( conteneurLoader );

      }

      function clicImages ( pEvt:MouseEvent ):void

      {

        // [MouseEvent type="click" bubbles=true cancelable=false eventPhase=1
      localX=10 localY=49 stageX=250 stageY=49 relatedObject=null ctrlKey=false
      altKey=false shiftKey=false delta=0]
        trace( pEvt );

      }
```

L'activation du mode bouton est rendue possible grâce à l'activation de la propriété `buttonMode` sur les instances de `Sprite` contenant les objets `Loader` :

```
        conteneurLoader.buttonMode = true;
```

Puis nous modifions la fonction `clicImages` :

```
      function clicImages ( pEvt:MouseEvent ):void

      {

        DisplayObject( pEvt.target ).alpha = .5;

      }
```

La figure 13-4 illustre le résultat :

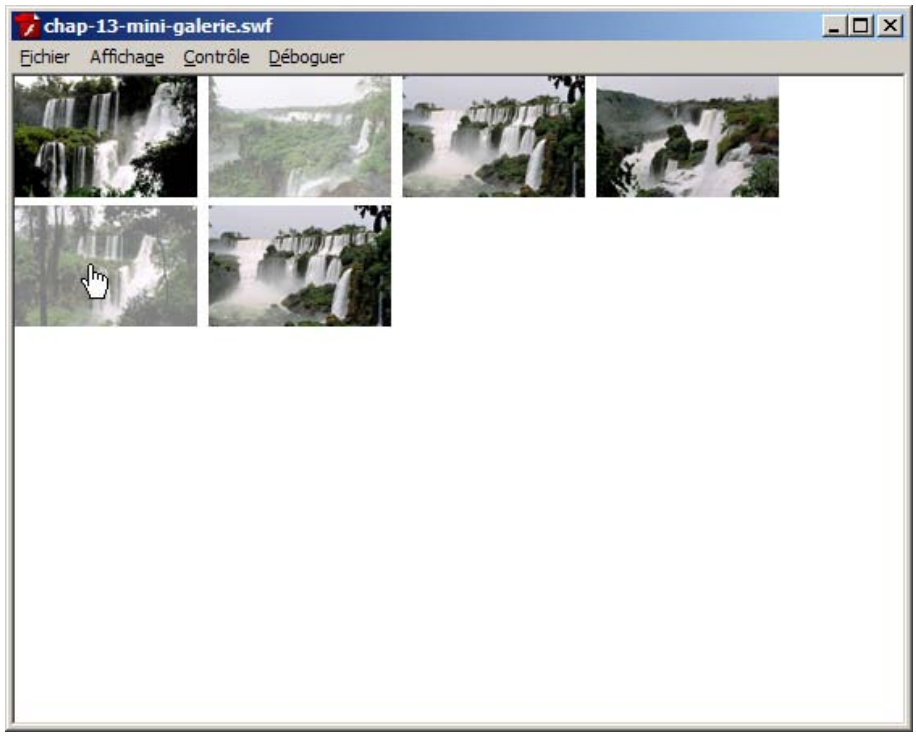

Figure 13-4. Mini galerie.

A chaque image cliquée, celle-ci passe en opacité réduite afin de mémoriser visuellement l'action utilisateur.

A retenir

- Le contenu chargé est accessible par la propriété `content` de l'objet `LoaderInfo`.
- Bien qu'étant une sous-classe de `DisplayObjectContainer`, la classe `Loader` ne dispose pas de toutes les capacités propres à ce type.
- Il est techniquement possible de déplacer le contenu chargé et de réattribuer son parent, mais il est fortement recommandé de conserver le contenu au sein de l'objet `Loader`.
- Afin d'afficher le curseur main au survol d'un objet `Loader`, nous devons au préalable l'imbriquer au sein d'un conteneur de type `Sprite`.

Créer une galerie

Nous allons mettre en pratique un ensemble de notions abordées depuis le début de l'ouvrage ainsi que celles abordées en début de ce chapitre. Nous allons créer une galerie photo et ainsi voir comment la concevoir de manière optimisée.

Ouvrez un nouveau document Flash CS3, puis définissez une classe de document au sein d'un paquetage `org.bytearray.document` :

```
 package org.bytearray.document

{

  import org.bytearray.abstrait.ApplicationDefaut;

  public class Document extends ApplicationDefaut

  {

        public function Document ()

        {

              trace( this );

        }

  }

}
```

Cette classe étend la classe `ApplicationDefaut` créée au cours du chapitre 11 intitulé *Classe du document,* nous récupérons ainsi toutes les fonctionnalités définies par celle-ci.

Une fois créée, nous définissons celle-ci comme classe du document grâce au panneau approprié de l'inspecteur de propriétés :

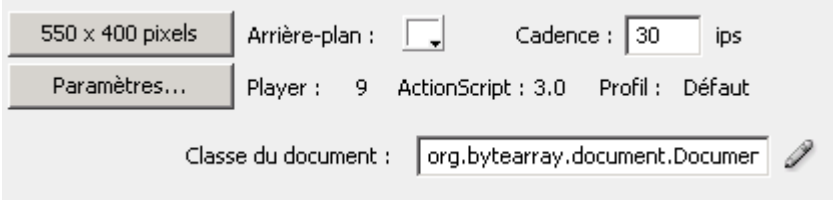

Figure 13-5. Classe du document.

Nous définissons la classe `ChargeurMedia` au sein du paquetage `org.bytearray.chargement`. Cette classe va nous permettre de gérer facilement le chargement d'images. Nous pourrions utiliser simplement la classe `Loader` mais nous avons besoin d'augmenter les ses fonctionnalités.

La classe `ChargeurMedia` se chargera de toute la logique de chargement et de redimensionnement des médias chargés. Ainsi pour chaque projet nécessitant un chargement de contenu notre classe `ChargeurMedia` pourra être réutilisée.

Voici le code de base de la classe `ChargeurMedia` :

```
package org.bytearray.chargement

{

  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.display.LoaderInfo;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;

  public class ChargeurMedia extends Sprite

  {

        private var chargeur:Loader;
        private var cli:LoaderInfo;

        public function ChargeurMedia ()

        {

              chargeur = new Loader();

              addChild ( chargeur );
```

```
            cli = chargeur.contentLoaderInfo;

            cli.addEventListener ( Event.INIT, initChargement );
            cli.addEventListener ( Event.OPEN, demarreChargement );
            cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
            cli.addEventListener ( Event.COMPLETE, chargementTermine );
            cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );

        }

        private function demarreChargement ( pEvt:Event ):void

        {

            trace ("démarre chargement");

        }

        private function initChargement ( pEvt:Event ):void

        {

            trace ("initialisation du chargement");

        }

        private function chargement ( pEvt:Event ):void

        {

            trace ("chargement en cours");

        }

        private function chargementTermine ( pEvt:Event ):void

        {

            trace ("chargement terminé");

        }

        private function echecChargement ( pEvt:Event ):void

        {

            trace ("erreur de chargement");

        }

    }
}
```

Une fois la classe `ChargeurMedia` définie, nous pouvons l'instancier au sein de la classe `Document` :

```
package org.bytearray.document

{

    import org.bytearray.abstrait.ApplicationDefaut;
```

```
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefaut

{

      // visionneuse
      private var visionneuse:ChargeurMedia;

      public function Document ()

      {

            // création de l'objet Loader
            visionneuse = new ChargeurMedia ();

            addChild ( visionneuse );

      }

}

}
```

Certains d'entre vous seront peut-être étonnés de voir que nous n'héritons pas de la classe `Loader`. Pourtant, toutes les raisons paraissent ici réunies pour utiliser l'héritage. Notre classe `ChargeurMedia` doit augmenter les capacités de la classe `Loader`, l'héritage semble donc être la meilleure solution.

Nous allons préférer ici une deuxième approche déjà utilisée au cours des chapitres précédents. La classe `ChargeurMedia` ne *sera* pas un objet `Loader` mais *possèdera* un objet `Loader`.

Vous souvenez-vous de cette opposition ?

La composition

Nous allons préférer ici l'utilisation de la *composition* à *l'héritage*. Nous avons vu au cours du chapitre 8 intitulé *Programmation orientée objet* les avantages qu'offre la composition en matière d'évolutivité et de modification du code.

Lorsque l'application est exécutée, le constructeur de la classe `Document` instancie un objet `ChargeurMedia`. Très généralement, les données à charger proviennent d'une base de données ou d'un fichier XML. Nous allons réaliser une première version à l'aide d'un tableau contenant les URL des images à charger.

Nous créons ensuite un tableau contenant l'adresse de chaque image :

```
package org.bytearray.document

{
```

```
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefaut

{

      // visionneuse
      private var visionneuse:ChargeurMedia;

      // données
      private var tableauImages:Array;

      public function Document ()

      {

            // tableau contenant les url des images
            tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

            // création de l'objet Loader
            visionneuse = new ChargeurMedia ();

            addChild ( visionneuse );

      }

}

}
```

Un répertoire `imgs` est créé contenant différentes images portant le nom de celles définies dans le tableau `tableauImages`. Pour le moment, la classe `ChargeurMedia` n'est pas utilisable, nous devons ajouter une méthode permettant de charger le media spécifié.

Nous importons les classes `URLRequest` et `LoaderContext` dans la définition de la classe :

```
import flash.display.Loader;
import flash.display.Sprite;
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.events.IOErrorEvent;
import flash.net.URLRequest;
import flash.system.LoaderContext;
```

Puis, nous ajoutons une méthode `load`. Celle-ci appelle en interne la méthode `load` de l'objet `Loader` :

```
public function load ( pURL:URLRequest, pContext:LoaderContext=null ):void

{

      chargeur.load ( pURL, pContext );

}
```

En délégant les fonctionnalités à une classe interne. La classe `ChargeurMedia` procède à une *délégation*.

> Ce mécanisme est l'un des plus puissants de la composition. Grâce à cela, la classe `ChargeurMedia` peut à l'exécution décider d'utiliser un autre type d'objet dans lequel charger les données.
>
> Chose impossible si la classe `ChargeurMedia` avait héritée de la classe `Loader`. C'est pour cette raison que la composition est considérée comme une approche dynamique, s'opposant au caractère statique de l'héritage lié à la compilation.

Sur la scène nous posons deux boutons `boutonSuivant` et `boutonPrecedent`, auxquels nous donnons deux noms d'occurrences respectifs.

La figure 13-6 illustre l'interface :

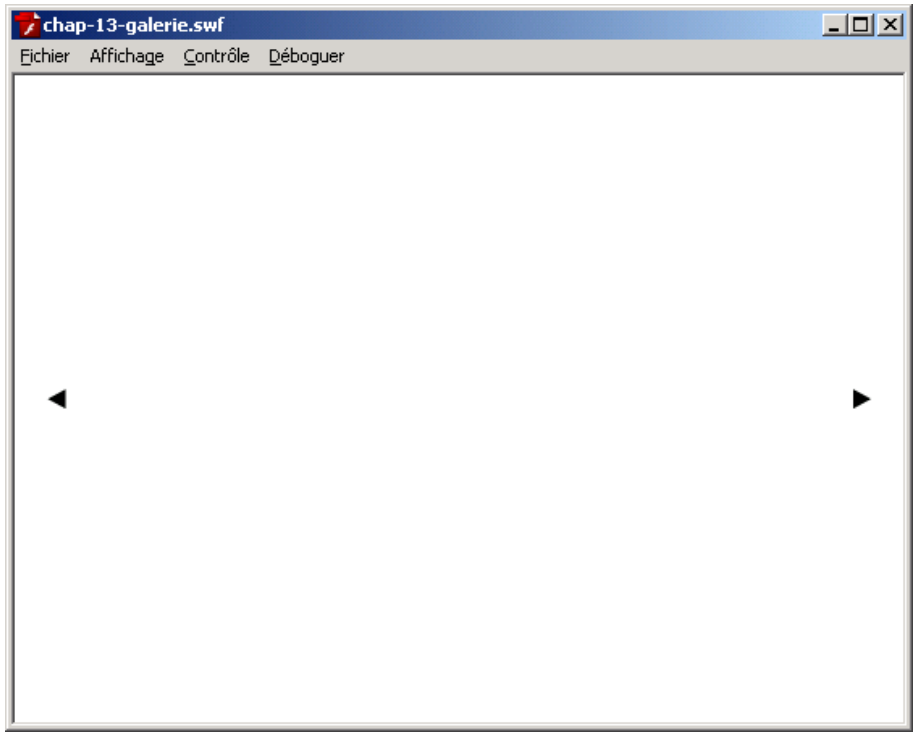

Figure 13-6. Boutons de navigation.

Nous définissons chaque bouton au sein de la classe `Document` :

```
package org.bytearray.document

{
```

```
import flash.display.SimpleButton;
import flash.events.MouseEvent;
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefaut

{

      // interface
      public var boutonSuivant:SimpleButton;
      public var boutonPrecedent:SimpleButton;

      // visionneuse
      private var visionneuse:ChargeurMedia;

      // données
      private var tableauImages:Array;

      public function Document ()

      {

            // tableau contenant les url des images
            tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

            // création de l'objet Loader
            visionneuse = new ChargeurMedia ();

            addChild ( visionneuse );

            boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
            boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

      }

      private function clicPrecedent ( pEvt:MouseEvent=null ):void

      {

            trace("Image précédente");

      }

      private function clicSuivant ( pEvt:MouseEvent=null ):void

      {

            trace("Image suivante");

      }

}
}
```

Puis nous associons la logique spécifique à chaque bouton :

```
package org.bytearray.document
```

```
{

  import flash.display.SimpleButton;
  import flash.events.MouseEvent;
  import flash.net.URLRequest;
  import org.bytearray.abstrait.ApplicationDefaut;
  import org.bytearray.chargement.ChargeurMedia;

  public class Document extends ApplicationDefaut

  {

        // interface
        public var boutonSuivant:SimpleButton;
        public var boutonPrecedent:SimpleButton;

        // visionneuse
        private var visionneuse:ChargeurMedia;

        // données
        private var tableauImages:Array;
        private var position:int;
        private var requete:URLRequest;

        public function Document ()

        {

              position = -1;

              requete = new URLRequest();

              // tableau contenant les url des images
              tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

              // création de l'objet Loader
              visionneuse = new ChargeurMedia ();

              addChild ( visionneuse );

              boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
              boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

        }

        private function clicPrecedent ( pEvt:MouseEvent=null ):void

        {

              position = Math.max ( 0, --position );

              requete.url = tableauImages [ position ];

              visionneuse.load ( requete );

        }

        private function clicSuivant ( pEvt:MouseEvent=null ):void

        {
```

```
            position = ++position % tableauImages.length;
            requete.url = tableauImages [ position ];
            visionneuse.load ( requete );
        }
    }
}
```

A chaque clic, nous déclenchons la fonction `clicPrecedent` ou `clicSuivant`. Nous remarquons que lorsque nous cliquons sur le bouton `boutonSuivant` nous tournons en boucle, à l'inverse le `boutonPrecedent` bute à l'index 0.

En créant une propriété `position`, nous incrémentons ou décrémentons celle-ci, afin de naviguer au sein du tableau `tableauImages`.

Ne vous est-il jamais arrivé de charger des images de différentes dimensions, et de souhaiter que quelles que soient leurs tailles, celles-ci s'affiche correctement dans un espace donné ?

Redimensionnement automatique

L'une des premières fonctionnalités que nous allons ajouter à la classe `Loader` concerne l'ajustement automatique de l'image selon une dimension spécifiée.

Les images chargées au sein de la galerie peuvent parfois déborder ou ne pas être ajustées automatiquement. Cela est généralement prévu par la partie serveur mais afin de sécuriser notre application, nous allons intégrer ce mécanisme côté client. Cela nous évitera de mauvaises surprises au cas où l'image n'aurait pas été correctement redimensionnée au préalable.

Lorsque l'événement `Event.COMPLETE` est diffusé, nous ajustons automatiquement la taille de l'objet `ChargeurMedia` aux dimensions voulues tout en conservant les proportions afin de ne pas déformer le contenu chargé :

```
package org.bytearray.chargement

{

  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.display.LoaderInfo;
  import flash.events.Event;
  import flash.events.ProgressEvent;
```

```
import flash.events.IOErrorEvent;
import flash.net.URLRequest;
import flash.system.LoaderContext;
import flash.display.DisplayObject;

public class ChargeurMedia extends Sprite

{

	private var chargeur:Loader;
	private var cli:LoaderInfo;
	private var largeur:Number;
	private var hauteur:Number;

	public function ChargeurMedia ( pLargeur:Number, pHauteur:Number )

	{

		largeur = pLargeur;

		hauteur = pHauteur;

		chargeur = new Loader();

		addChild ( chargeur ) ;

		cli = chargeur.contentLoaderInfo;

		cli.addEventListener ( Event.INIT, initChargement );
		cli.addEventListener ( Event.OPEN, demarreChargement );
		cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
		cli.addEventListener ( Event.COMPLETE, chargementTermine );
		cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );

	}

	public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void

	{

		chargeur.load ( pURL, pContext );

	}

	private function demarreChargement ( pEvt:Event ):void

	{

		trace ("démarre chargement");

	}

	private function initChargement ( pEvt:Event ):void

	{

		trace ("initialisation chargement");

	}

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

```
        {
            trace ("chargement en cours");
        }
        private function chargementTermine ( pEvt:Event ):void
        {
            var contenuCharge:DisplayObject = pEvt.target.content;
            var ratio:Number = Math.min (  largeur / contenuCharge.width,
hauteur / contenuCharge.height );
            scaleX = scaleY = 1;
            if ( ratio < 1 ) scaleX = scaleY = ratio;
        }
        private function echecChargement ( pEvt:Event ):void
        {
            trace ("erreur de chargement");
        }
  }
}
```

Puis nous instancions l'objet `ChargeurMedia` en spécifiant les dimensions à respecter :

```
// création de la visionneuse
visionneuse = new ChargeurMedia ( 350, 450 );
```

Si nous testons la galerie nous remarquons que les images sont bien affichées et correctement redimensionnées si cela est nécessaire.

Pour le moment, elles ne sont pas centrées :

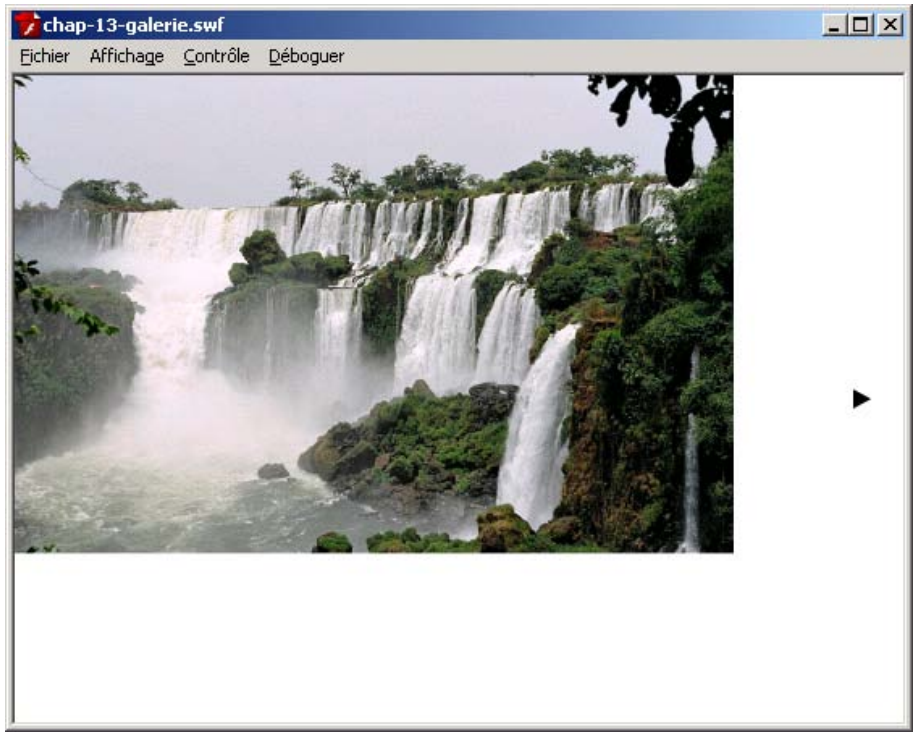

Figure 13-7. Galerie.

Nous devons placer le code correspondant au centrage de l'image en dehors de la classe `ChargeurMedia`. Intégrer une telle logique à notre classe `ChargeurMedia` la rendrait rigide.

N'oubliez pas un point essentiel de la programmation orientée objet ! Nous devons isoler ce qui risque d'être modifié d'un projet à un autre, nous plaçons donc le positionnement des images en dehors de la classe `ChargeurMedia`.

Afin de pouvoir utiliser la classe `ChargeurMedia` comme la classe `Loader`, nous devons à présent diffuser tous les événements nécessaires à son bon fonctionnement. Afin de permettre cette diffusion nous devons d'abord les écouter en interne, puis les rediriger par la suite.

Nous modifions pour cela chaque fonction écouteur, puis nous procédons à la redirection de chaque événement :

```
package org.bytearray.chargement

{

  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.display.DisplayObject;
  import flash.display.LoaderInfo;
  import flash.events.Event;
```

```
import flash.events.ProgressEvent;
import flash.events.IOErrorEvent;
import flash.net.URLRequest;
import flash.system.LoaderContext;

public class ChargeurMedia extends Sprite

{

	private var chargeur:Loader;
	private var cli:LoaderInfo;
	private var largeur:Number;
	private var hauteur:Number;

	public function ChargeurMedia ( pLargeur:Number, pHauteur:Number )

	{

		largeur = pLargeur;

		hauteur = pHauteur;

		chargeur = new Loader();

		addChild ( chargeur ) ;

		cli = chargeur.contentLoaderInfo;

		cli.addEventListener ( Event.INIT, initChargement );
		cli.addEventListener ( Event.OPEN, demarreChargement );
		cli.addEventListener ( ProgressEvent.PROGRESS, chargement );
		cli.addEventListener ( Event.COMPLETE, chargementTermine );
		cli.addEventListener ( IOErrorEvent.IO_ERROR, echecChargement );

	}

	public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void

	{

		chargeur.load ( pURL, pContext );

	}

	private function demarreChargement ( pEvt:Event ):void

	{

		dispatchEvent ( pEvt );

	}

	private function initChargement ( pEvt:Event ):void

	{

		dispatchEvent ( pEvt );

	}

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

```
            {
                dispatchEvent ( pEvt );
            }
            private function chargementTermine ( pEvt:Event ):void
            {
                var contenuCharge:DisplayObject = pEvt.target.content;
                var ratio:Number = Math.min (  largeur / contenuCharge.width,
    hauteur / contenuCharge.height );
                scaleX = scaleY = 1;
                if ( ratio < 1 ) scaleX = scaleY = ratio;
                dispatchEvent ( pEvt );
            }
            private function echecChargement ( pEvt:Event ):void
            {
                dispatchEvent ( pEvt );
            }
      }
    }
```

De cette manière, la classe `ChargeurMedia`, diffuse les mêmes événements que la classe `Loader`. Cela reste totalement transparent pour l'utilisateur de la classe `ChargeurMedia` et s'inscrit comme suite logique de la *composition* et de la *délégation*.

> Lorsqu'ils sont redirigés, les objets événementiels voient leur propriété `target` référencer l'objet diffuseur ayant relayé l'événement. La propriété `target` fait donc désormais référence à l'objet `ChargeurMedia` et non plus à l'objet `LoaderInfo`.

En écoutant l'événement `Event.COMPLETE` auprès de la classe `ChargeurMedia`, nous sommes avertis de la fin du chargement du contenu :

```
package org.bytearray.document

{

  import flash.display.SimpleButton;
  import flash.events.Event;
  import flash.events.MouseEvent;
```

```
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefaut

{

      // interface
      public var boutonSuivant:SimpleButton;
      public var boutonPrecedent:SimpleButton;

      // visionneuse
      private var visionneuse:ChargeurMedia;

      public function Document ()

      {

            // tableau contenant les url des images
            var tableauImages:Array = new Array ( "imgs/photo1.jpg",
"imgs/anim1.swf", "imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

            // création de l'objet Loader
            visionneuse = new ChargeurMedia ( 350, 450 );

            addChild ( visionneuse );

            // écoute de l'événement Event.COMPLETE auprès de la visionneuse
            visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
);

            boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
            boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

      }

      private function chargementTermine ( pEvt:Event ):void

      {

            // centre la visionneuse
            visionneuse.x = ( stage.stageWidth - visionneuse.width ) / 2;
            visionneuse.y = ( stage.stageHeight - visionneuse.height ) / 2;

      }

      private function clicPrecedent ( pEvt:MouseEvent=null ):void

      {

            position = Math.max ( 0, --position );

            requete.url = tableauImages [ position ];

            visionneuse.load ( requete );

      }

      private function clicSuivant ( pEvt:MouseEvent=null ):void

      {
```

```
                position = ++position % tableauImages.length;
                requete.url = tableauImages [ position ];
                visionneuse.load ( requete );
            }
        }
    }
```

Afin de garantir un affichage aligné sur les pixels, il est recommandé de toujours positionner une image sur des coordonnées non flottantes.

Pour cela, nous modifions la fonction écouteur `chargementTermine` en arrondissant les coordonnées de l'image centrée :

```
private function chargementTermine ( pEvt:Event ):void
{
  // centre la visionneuse
  visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) / 2);
  visionneuse.y = int ( ( stage.stageHeight - visionneuse.height ) / 2);
}
```

Si nous testons la galerie, les images sont correctement chargées et centrées, comme l'illustre la figure 13-8 :

Figure 13-8. Galerie centrée.

Afin de charger la première image, nous pouvons déclencher manuellement la fonction écouteur `clicSuivant` :

```
public function Document ()

{

	// tableau contenant les url des images
	var tableauImages:Array = new Array ( "imgs/photo1.jpg",
"imgs/photo2.jpg", "imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

	// création de l'objet Loader
	visionneuse = new ChargeurMedia ( tableauImages, 350, 450 );

	addChild ( visionneuse );

	// écoute de l'événement Event.COMPLETE auprès de la visionneuse
	visionneuse.addEventListener ( Event.COMPLETE, chargementTermine );

	boutonPrecedent.addEventListener ( MouseEvent.CLICK, clicPrecedent );
	boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

	clicSuivant();

}
```

Lorsque l'application est lancée, la première image est chargée automatiquement. Nous venons de déclencher ici, manuellement une fonction écouteur liée à l'événement `MouseEvent.CLICK`.

Le code de la classe `ChargeurMedia` peut être optimisé en utilisant une seule fonction écouteur pour rediriger différents types d'événements.

Dans le code suivant la méthode `redirigeEvenement` redirige plusieurs événements :

```
package org.bytearray.chargement

{

  import flash.display.Bitmap;
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.display.DisplayObject;
  import flash.display.LoaderInfo;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;

  public class ChargeurMedia extends Sprite

  {

	private var chargeur:Loader;
```

```
		private var cli:LoaderInfo;
		private var largeur:Number;
		private var hauteur:Number;

		public function ChargeurMedia ( pLargeur:Number, pHauteur:Number )

		{

			largeur = pLargeur;

			hauteur = pHauteur;

			chargeur = new Loader();

			addChild ( chargeur ) ;

			cli = chargeur.contentLoaderInfo;

			cli.addEventListener ( Event.INIT, redirigeEvenement );
			cli.addEventListener ( Event.OPEN, redirigeEvenement );
			cli.addEventListener ( ProgressEvent.PROGRESS, redirigeEvenement
);
			cli.addEventListener ( Event.COMPLETE, chargementTermine );
			cli.addEventListener ( IOErrorEvent.IO_ERROR, redirigeEvenement
);

		}

		public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void

		{

			chargeur.load ( pURL, pContext );

		}

		private function redirigeEvenement ( pEvt:Event ):void

		{

			dispatchEvent ( pEvt );

		}

		private function chargementTermine ( pEvt:Event ):void

		{

			var contenuCharge:DisplayObject = pEvt.target.content;

			var ratio:Number = Math.min ( largeur / contenuCharge.width,
	hauteur / contenuCharge.height );

			scaleX = scaleY = 1;

			if ( ratio < 1 ) scaleX = scaleY = ratio;

			dispatchEvent ( pEvt );

		}
```

```
  }

}
```

Bien entendu, cette technique n'est viable que dans le cas où aucune logique ne doit être ajoutée en interne pour chaque événement diffusé. C'est pour cette raison que l'événement `Event.COMPLETE` est toujours écouté en interne par la méthode écouteur `chargementTermine`, car nous devons ajouter une logique de redimensionnement spécifique.

Gestion du lissage

Nous allons ajouter une seconde fonctionnalité permettant d'afficher les images chargées de manière lissée. Ainsi, lorsque l'image chargée est redimensionnée, grâce au lissage celle-ci n'est pas crénelée.

Pour cela, nous activons la propriété `smoothing` si le contenu chargé est de type `Bitmap` :

```
package org.bytearray.chargement

{

  import flash.display.Bitmap;
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.display.DisplayObject;
  import flash.display.LoaderInfo;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;

  public class ChargeurMedia extends Sprite

  {

        private var chargeur:Loader
        private var cli:LoaderInfo;
        private var largeur:Number;
        private var hauteur:Number;
        private var lissage:Boolean;

        public function ChargeurMedia ( pLargeur:Number, pHauteur:Number,
pLissage:Boolean=true )

        {

              largeur = pLargeur;

              hauteur = pHauteur;

              lissage = pLissage;

              chargeur = new Loader();
```

```
                addChild ( chargeur ) ;

                cli = chargeur.contentLoaderInfo;

                cli.addEventListener ( Event.INIT, redirigeEvenement );
                cli.addEventListener ( Event.OPEN, redirigeEvenement );
                cli.addEventListener ( ProgressEvent.PROGRESS, redirigeEvenement
);
                cli.addEventListener ( Event.COMPLETE, chargementTermine );
                cli.addEventListener ( IOErrorEvent.IO_ERROR, redirigeEvenement
);

        }

        public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void

        {

                chargeur.load ( pURL, pContext );

        }

        private function redirigeEvenement ( pEvt:Event ):void

        {

                dispatchEvent ( pEvt );

        }

        private function chargementTermine ( pEvt:Event ):void

        {

                var contenuCharge:DisplayObject = pEvt.target.content;

                if ( contenuCharge is Bitmap ) Bitmap ( contenuCharge ).smoothing
= lissage;

                var ratio:Number = Math.min ( largeur / contenuCharge.width,
hauteur / contenuCharge.height );

                scaleX = scaleY = 1;

                if ( ratio < 1 ) scaleX = scaleY = ratio;

                dispatchEvent ( pEvt );

        }

    }

}
```

Puis nous modifions l'instanciation de l'objet `ChargeurMedia` :

```
// création de l'objet Loader
visionneuse = new ChargeurMedia ( tableauImages, 350, 450, true );
```

Les images sont désormais lissées automatiquement. Si nous souhaitons désactiver le lissage, nous le précisons lors de l'instanciation de l'objet `ChargeurMedia` :

```
// création de l'objet Loader
visionneuse = new ChargeurMedia ( tableauImages, 350, 450, false );
```

Souvenez-vous, nous devons toujours intégrer un mécanisme de `desactivation` des objets. Ainsi la classe `ChargeurMedia` ne déroge pas à la règle et intègre un mécanisme de désactivation. Aussitôt supprimée de la liste d'affichage, l'instance de `ChargeurMedia` est désactivée grâce à l'événement `Event.REMOVED_FROM_STAGE`.

Nous utilisons l'événement `Event.ADDED_TO_STAGE` afin d'initialiser l'objet `ChargeurMedia` lorsque celui-ci est affiché :

```
package org.bytearray.chargement

{

  import flash.display.Bitmap;
  import flash.display.Loader;
  import flash.display.Sprite;
  import flash.display.DisplayObject;
  import flash.display.LoaderInfo;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.events.IOErrorEvent;
  import flash.net.URLRequest;
  import flash.system.LoaderContext;

  public class ChargeurMedia extends Sprite

  {

        private var chargeur:Loader;
        private var cli:LoaderInfo;
        private var largeur:Number;
        private var hauteur:Number;
        private var lissage:Boolean;

        public function ChargeurMedia ( pLargeur:Number, pHauteur:Number,
pLissage:Boolean=true )

        {

              largeur = pLargeur;

              hauteur = pHauteur;

              lissage = pLissage;

              chargeur = new Loader();

              addChild ( chargeur );

              cli = chargeur.contentLoaderInfo;
```

```
            addEventListener ( Event.ADDED_TO_STAGE, activation );
            addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

        }

        private function activation ( pEvt:Event ):void

        {

            cli.addEventListener ( Event.INIT, redirigeEvenement );
            cli.addEventListener ( Event.OPEN, redirigeEvenement );
            cli.addEventListener ( ProgressEvent.PROGRESS, redirigeEvenement
);
            cli.addEventListener ( Event.COMPLETE, chargementTermine );
            cli.addEventListener ( IOErrorEvent.IO_ERROR, redirigeEvenement
);

        }

        private function desactivation ( pEvt:Event ):void

        {

            cli.removeEventListener ( Event.INIT, redirigeEvenement );
            cli.removeEventListener ( Event.OPEN, redirigeEvenement );
            cli.removeEventListener ( ProgressEvent.PROGRESS,
redirigeEvenement );
            cli.removeEventListener ( Event.COMPLETE, chargementTermine );
            cli.removeEventListener ( IOErrorEvent.IO_ERROR,
redirigeEvenement );

        }

        public function load ( pURL:URLRequest, pContext:LoaderContext=null
):void

        {

            chargeur.load ( pURL, pContext );

        }

        private function redirigeEvenement ( pEvt:Event ):void

        {

            dispatchEvent ( pEvt );

        }

        private function chargementTermine ( pEvt:Event ):void

        {

            var contenuCharge:DisplayObject = pEvt.target.content;

            if ( contenuCharge is Bitmap ) Bitmap ( contenuCharge ).smoothing
= lissage;

            var ratio:Number = Math.min ( largeur / contenuCharge.width,
hauteur / contenuCharge.height );
```

```
            scaleX = scaleY = 1;
            if ( ratio < 1 ) scaleX = scaleY = ratio;
            dispatchEvent ( pEvt );
        }
    }
}
```

La classe `ChargeurMedia` fonctionne sans problème. Nous verrons au cours du chapitre 14 intitulé *Chargement de données* comment remplacer les données par un flux XML chargé dynamiquement.

Nous verrons comment réutiliser la classe `ChargeurMedia` tout en la faisant communiquer avec une classe permettant le chargement de données XML externe.

Précharger le contenu

Il est impératif de précharger tout contenu dynamique. Un utilisateur interrompt très rapidement sa navigation si aucun élément n'indique visuellement l'état du chargement des données.

Dans le cas de notre galerie photo, nous devons précharger chaque image afin d'informer l'utilisateur de l'état du chargement. Nous allons utiliser une barre de préchargement classique puis utiliser chacun des événements afin de synchroniser son affichage.

Nous allons tout d'abord créer un symbole contenant la barre de préchargement. La figure 13-9 illustre le symbole en bibliothèque :

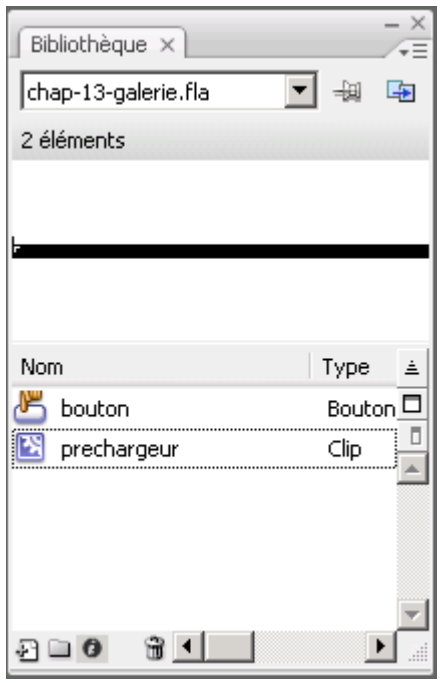

Figure 13-9. Symbole prechargeur.

Puis nous définissons une classe `Prechargeur` auquel le symbole est lié :

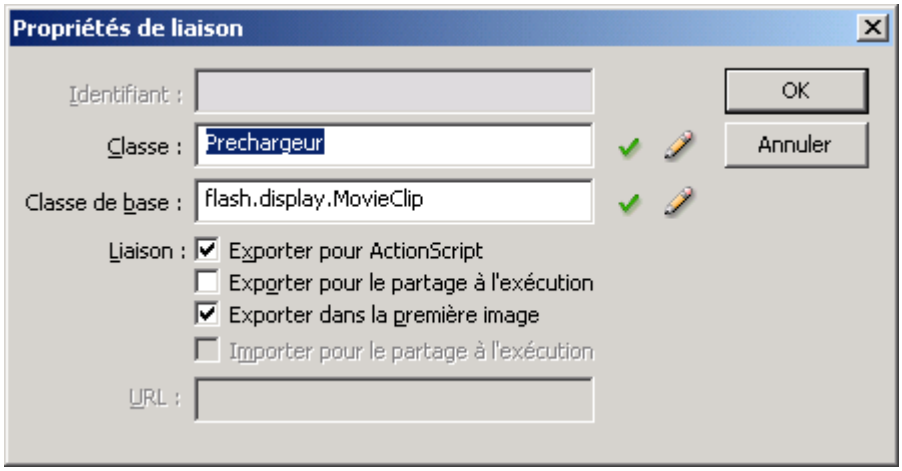

Figure 13-10. Classe `Prechargeur` associée.

Une fois le symbole associé, nous l'instancions dès l'initialisation de l'application. Vous remarquerez que celui-ci n'est pas affiché pour l'instant :

```
package org.bytearray.document

{

  import flash.display.SimpleButton;
  import flash.display.Sprite;
```

```
import flash.events.MouseEvent;
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.net.URLRequest;
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefaut

{

      // interface
      public var boutonSuivant:SimpleButton;
      public var boutonPrecedent:SimpleButton;
      public var prechargeur:Prechargeur;

      // visionneuse
      private var visionneuse:ChargeurMedia;

      // données
      private var tableauImages:Array;
      private var position:int;
      private var requete:URLRequest;

      public function Document ()

      {

            position = -1;

            requete = new URLRequest();

            // tableau contenant les url des images
            tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

            // création de l'objet Loader
            visionneuse = new ChargeurMedia ( 445, 335 );

            addChild ( visionneuse );

            prechargeur = new Prechargeur();

            prechargeur.x = Math.round ( (stage.stageWidth -
prechargeur.width) / 2);
            prechargeur.y = Math.round ( (stage.stageHeight -
prechargeur.height) / 2);

            visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
);

            boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
            boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

            clicSuivant();

      }

      private function chargementTermine ( pEvt:Event ):void

      {
```

```
            // centre la visionneuse
            visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) /
2);
            visionneuse.y = int ( ( stage.stageHeight - visionneuse.height )
/ 2);

      }

      private function clicPrecedent ( pEvt:MouseEvent=null ):void

      {

            position = Math.max ( 0, --position );

            requete.url = tableauImages [ position ];

            visionneuse.load ( requete );

      }

      private function clicSuivant ( pEvt:MouseEvent=null ):void

      {

            position = ++position % tableauImages.length;

            requete.url = tableauImages [ position ];

            visionneuse.load ( requete );

      }

  }

}
```

Puis nous écoutons chacun des événements nécessaires au préchargement :

```
package org.bytearray.document

{

  import flash.display.SimpleButton;
  import flash.display.Sprite;
  import flash.events.MouseEvent;
  import flash.events.Event;
  import flash.events.ProgressEvent;
  import flash.net.URLRequest;
  import org.bytearray.abstrait.ApplicationDefaut;
  import org.bytearray.chargement.ChargeurMedia;

  public class Document extends ApplicationDefaut

  {

        // interface
        public var boutonSuivant:SimpleButton;
        public var boutonPrecedent:SimpleButton;
        public var prechargeur:Prechargeur;
```

```
        // visionneuse
        private var visionneuse:ChargeurMedia;

        // données
        private var tableauImages:Array;
        private var position:int;
        private var requete:URLRequest;

        public function Document ()

        {

             position = -1;

             requete = new URLRequest();

             // tableau contenant les url des images
             tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

             // création de l'objet Loader
             visionneuse = new ChargeurMedia ( 445, 335 );

             addChild ( visionneuse );

             prechargeur = new Prechargeur();

             prechargeur.x = Math.round ( (stage.stageWidth -
prechargeur.width) / 2);
             prechargeur.y = Math.round ( (stage.stageHeight -
prechargeur.height) / 2);

             visionneuse.addEventListener ( Event.OPEN, chargementDemarre );
             visionneuse.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
             visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
);

             boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
             boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

             clicSuivant ( new MouseEvent ( MouseEvent.CLICK ) );

        }

        private function chargementDemarre ( pEvt:Event ):void

        {


        }

        private function chargementEnCours ( pEvt:ProgressEvent ):void

        {


        }

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

```
            {

                  // centre la visionneuse
                  visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) /
2);
                  visionneuse.y = int ( ( stage.stageHeight - visionneuse.height )
/ 2);

            }

            private function clicPrecedent ( pEvt:MouseEvent=null ):void

            {

                  position = Math.max ( 0, --position );

                  requete.url = tableauImages [ position ];

                  visionneuse.load ( requete );

            }

            private function clicSuivant ( pEvt:MouseEvent=null ):void

            {

                  position = ++position % tableauImages.length;

                  requete.url = tableauImages [ position ];

                  visionneuse.load ( requete );

            }

        }

      }
```

Lorsque le chargement démarre, nous ajoutons la barre de préchargement à l'affichage si celle-ci n'est pas déjà affichée :

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

      {

            if ( !contains ( prechargeur ) ) addChild ( prechargeur );

      }
```

Puis nous adaptons sa taille selon le pourcentage chargé :

```
      private function chargementEnCours ( pEvt:ProgressEvent ):void

      {

            prechargeur.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;

      }
```

Enfin, nous supprimons la barre de préchargement lorsque les données sont chargées :

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

{

	// centre la visionneuse
	visionneuse.x = int ( ( stage.stageWidth - visionneuse.width ) / 2);
	visionneuse.y = int ( ( stage.stageHeight - visionneuse.height ) / 2);

	if ( contains ( prechargeur ) ) removeChild ( prechargeur );

}
```

En testant notre galerie, chaque image est préchargée, comme l'illustre la figure 13-11 :

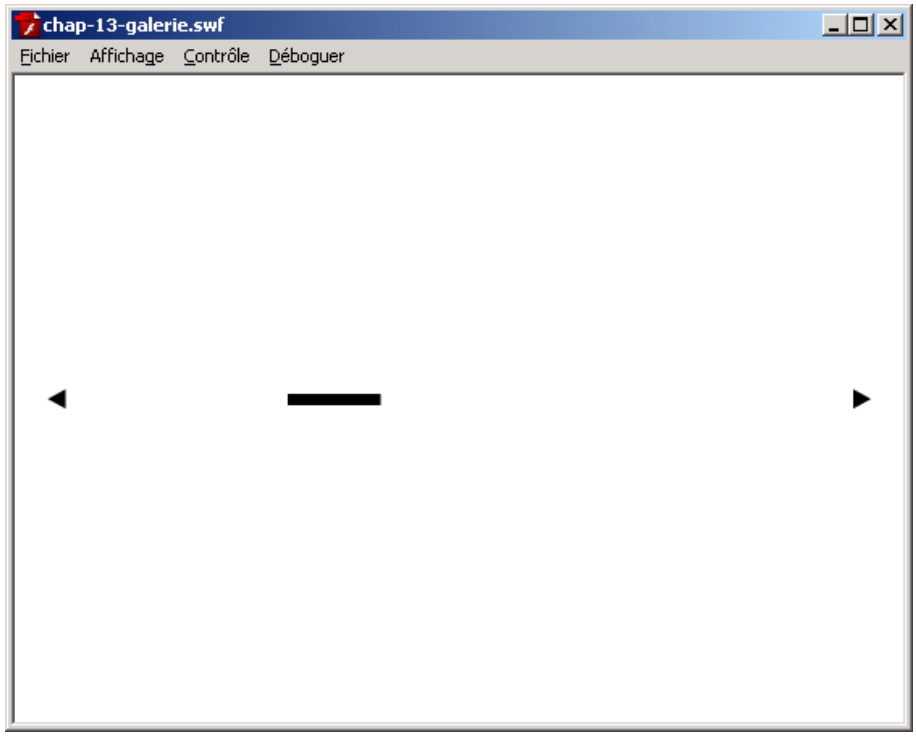

Figure 13-11. Préchargement des images.

Libre à vous d'intégrer différents types d'éléments indiquant le niveau de chargement des données. Nous aurions pu utiliser une barre de préchargement accompagnée d'un champ texte affichant un pourcentage de 0 à 100, ou bien une animation se jouant selon le volume de données chargées.

Afin d'ajouter un peu d'esthétisme à notre galerie, nous ajoutons un effet de fondu permettant aux photos d'être affichées progressivement :

```
package org.bytearray.document

{

  import flash.display.SimpleButton;
```

```
import flash.display.Sprite;
import flash.events.MouseEvent;
import flash.events.Event;
import flash.events.ProgressEvent;
import flash.net.URLRequest;
import fl.transitions.Tween;
import fl.transitions.easing.Strong;
import org.bytearray.abstrait.ApplicationDefaut;
import org.bytearray.chargement.ChargeurMedia;

public class Document extends ApplicationDefaut

{

      // interface
      public var boutonSuivant:SimpleButton;
      public var boutonPrecedent:SimpleButton;
      public var prechargeur:Prechargeur;

      // visionneuse
      private var visionneuse:ChargeurMedia;

      // données
      private var tableauImages:Array;
      private var position:int;
      private var requete:URLRequest;

      // fondu
      private var fondu:Tween;

      public function Document ()

      {

            position = -1;

            requete = new URLRequest();

            // tableau contenant les url des images
            tableauImages = new Array ( "imgs/photo1.jpg", "imgs/photo2.jpg",
"imgs/photo3.jpg", "imgs/photo4.jpg", "imgs/photo5.jpg" );

            // création de l'objet Loader
            visionneuse = new ChargeurMedia ( 445, 335 );

            addChild ( visionneuse );

            fondu = new Tween ( visionneuse, "alpha", Strong.easeOut, 1, 1,
0, true );

            prechargeur = new Prechargeur();

            prechargeur.x = int ( (stage.stageWidth - prechargeur.width) /
2);
            prechargeur.y = int ( (stage.stageHeight - prechargeur.height) /
2);

            visionneuse.addEventListener ( Event.OPEN, chargementDemarre );
            visionneuse.addEventListener ( ProgressEvent.PROGRESS,
chargementEnCours );
            visionneuse.addEventListener ( Event.COMPLETE, chargementTermine
);
```

```
            boutonPrecedent.addEventListener ( MouseEvent.CLICK,
clicPrecedent );
            boutonSuivant.addEventListener ( MouseEvent.CLICK, clicSuivant );

            clicSuivant ( new MouseEvent ( MouseEvent.CLICK ) );

      }

      private function chargementDemarre ( pEvt:Event ):void

      {

            if ( !contains ( prechargeur ) ) addChild ( prechargeur );

      }

      private function chargementEnCours ( pEvt:ProgressEvent ):void

      {

            prechargeur.scaleX = pEvt.bytesLoaded / pEvt.bytesTotal;

      }

      private function chargementTermine ( pEvt:Event ):void

      {

            // centre la visionneuse
            visionneuse.x = Math.round ( ( stage.stageWidth -
visionneuse.width ) / 2);
            visionneuse.y = Math.round ( ( stage.stageHeight -
visionneuse.height ) / 2);

            if ( contains ( prechargeur ) ) removeChild ( prechargeur );

            fondu.continueTo ( 1, 1 );

      }

      private function clicPrecedent ( pEvt:MouseEvent=null ):void

      {

            fondu.continueTo ( 0, 1 );

            position = Math.max ( 0, --position );

            requete.url = tableauImages [ position ];

            visionneuse.load ( requete );

      }

      private function clicSuivant ( pEvt:MouseEvent=null ):void

      {

            fondu.continueTo ( 0, 1 );

            position = ++position % tableauImages.length;
```

```
                requete.url = tableauImages [ position ];
                visionneuse.load ( requete );
            }
        }
    }
```

En testant notre galerie, nous remarquons un effet de fondu entre chaque image. Nous pouvons rendre le type de transition dynamique en passant en paramètre un type de comportement spécifique.

A vous d'imaginer la suite, pour cela voici un indice :

En privilégiant une approche par composition, une classe comportementale pourrait être définie puis passée en paramètre à la classe `ChargeurMedia` afin de terminer un type de transition.

A retenir

- Le contenu doit toujours être préchargé afin d'offrir une expérience utilisateur optimale.

Interrompre le chargement

Il était impossible avec les précédentes versions d'ActionScript d'interrompre le chargement d'un élément externe. Le seul moyen était de fermer le lecteur Flash. ActionScript 3 intègre dorénavant une méthode `close` sur la classe `Loader` permettant de stopper instantanément le chargement d'un élément.

Afin de rendre notre classe `ChargeurMedia` souple, nous pouvons ajouter une nouvelle méthode `close` :

```
public function close ( ):void
{
        chargeur.close();
}
```

Celle-ci délègue cette fonctionnalité à la classe `Loader` interne. Lorsque la méthode `load` est appelée, tout chargement précédemment initié est interrompu.

Nous n'avons pas abordé jusqu'à présent la notion de communication entre l'objet `Loader` et le contenu. Dans la partie suivante nous allons nous attarder sur la communication entre deux animations.

A retenir

- La méthode `close` de l'objet `Loader` permet d'interrompre le chargement en cours.

Communiquer entre deux animations

Dans un nouveau document Flash CS3 nous plaçons sur la scène une instance de symbole animé. L'animation est représentée par une instance du symbole `Garcon` utilisé lors du chapitre précédent. Nous lui donnons `animation` comme nom d'occurrence :

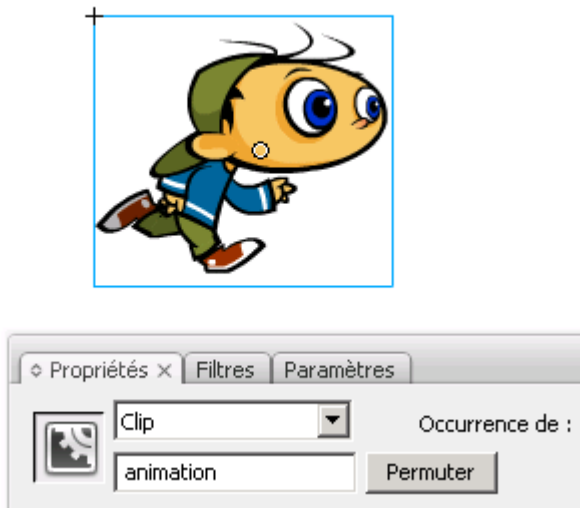

Figure 13-12. Instance du symbole `Garcon`.

Une fois l'animation exportée, nous la chargeons à l'aide de l'objet `Loader` depuis un autre SWF :

```
var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine );

chargeur.load ( new URLRequest ("chap-13-anim.swf") );

addChild ( chargeur );

function termine ( pEvt:Event ):void

{

  // référence le scénario de l'animation chargée
  var scenario:DisplayObject = pEvt.target.content;

  // affiche : [object MainTimeline]
  trace( scenario );
```

```
}
```

Une fois l'animation chargée, nous pouvons stopper l'animation en utilisant la syntaxe pointée :

```
function termine ( pEvt:Event ):void

{

  // référence le scénario de l'animation chargée
  var scenario:DisplayObject = pEvt.target.content;

  // si le scénario est un MovieClip nous accédons
  // à l'animation et la stoppons
  if ( scenario is MovieClip ) MovieClip ( scenario ).animation.stop();

}
```

Si l'accès au contenu de l'animation chargée s'avère aisée, la communication inverse s'avère plus complexe.

Afin d'accéder au scénario du SWF initiant le chargement, nous utilisons la propriété `root` du scénario principal du SWF chargé, puis à la propriété `parent` de ce même scénario :

```
// affiche : [object Loader]
trace( root.parent );
```

Nous accédons ainsi à l'objet `Loader` chargeant actuellement notre animation. En ciblant la propriété `root` de ce même objet nous accédons au scénario principal du SWF chargeur :

```
// affiche : [object MainTimeline]
trace( root.parent.root );
```

Afin de cibler une animation posée sur ce même scénario, nous pouvons écrire le code suivant :

```
// référence le scénario principal du SWF parent
var scenarioPrincipal:DisplayObject = root.parent.root;

// si celui-ci est un MovieClip
if ( scenarioPrincipal is MovieClip )
{

  // alors nous transtypons et accédons au clip animation
  MovieClip ( scenarioPrincipal ).animation.stop();

}
```

Notons que si l'objet `Loader` initiant le chargement n'est pas présent au sein de la liste d'affichage, sa propriété `root` renverra `null`.

Dans l'exemple précédent, nous n'avons rencontré aucune difficulté à accéder au contenu l'animation chargée car les deux animations évoluent depuis le même domaine.

Une autre technique plus élégante consiste à diffuser un événement personnalisé depuis le SWF initiant le chargement. Le SWF chargé est à l'écoute de ce dernier et reçoit les informations de manière souple et élégante.

Pour réaliser cet échange, nous utilisons l'objet `EventDispatcher` disponible par la propriété `sharedEvents` de la classe `LoaderInfo`.

Au sein du SWF initiant le chargement, nous diffusons un événement personnalisé contenant les informations à transmettre :

```
var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine );

chargeur.load ( new URLRequest ("chap-13-anim.swf") );

addChild ( chargeur );

function termine ( pEvt:Event ):void

{

  // nous diffusons un événement EvenementInfos.INFOS au SWF chargé
  pEvt.target.sharedEvents.dispatchEvent ( new EvenementInfos (
EvenementInfos.INFOS, "contenu !" ) );

}
```

Au sein du SWF chargé nous écoutons ce même événement :

```
loaderInfo.sharedEvents.addEventListener ( EvenementInfos.INFOS, ecouteur );

function ecouteur ( pEvt:EvenementInfos ):void

{

// affiche : contenu !
  trace( pEvt.infos );

}
```

Cette approche facilite grandement la communication entre deux animations et doit être considérée en priorité car elle ne souffre d'aucune restriction de sécurité même en contexte interdomaine.

Dans un contexte de chargement de contenu externe, nous risquons d'être confrontés aux restrictions de sécurité du lecteur Flash. Dans la partie suivante nous allons faire le point sur ces limitations afin de mieux comprendre comment les appréhender et résoudre certains problèmes.

Modèle de sécurité du lecteur Flash

Depuis le lecteur Flash 6, des restrictions de sécurité ont été ajoutées au sein du lecteur Flash lors du chargement ou d'envoi de données afin de protéger les auteurs de contenu divers.

Le modèle de sécurité du lecteur Flash appelé *Security Sandbox* en Anglais distingue deux acteurs :

- Le créateur du contenu
- Le chargeur de contenu

Ce modèle s'applique à tout type de contenu chargé au sein du lecteur. Il faut comprendre que par principe, lorsqu'un contenu graphique est chargé depuis un autre domaine, celui-ci est en *lecture seule* au sein du lecteur. Ce dernier refuse de scripter ou de modifier tout contenu graphique provenant d'un autre domaine. On dit alors que les fichiers évoluent dans un contexte *interdomaine*.

Par le terme *scripter* nous entendons l'accès par ActionScript à des données contenues dans une autre animation. Adobe utilise le terme de *programmation croisée* pour exprimer ce mécanisme.

Par le terme de *modification*, nous entendons par exemple l'activation de la propriété `smoothing` sur une image bitmap chargée, ou encore la capture d'une vidéo sous forme bitmap à l'aide de la méthode `draw` de la classe `BitmapData`.

Afin de bien comprendre le modèle de sécurité, nous allons étudier différentes situations. Commençons par le cas de figure suivant :

Nous développons un portail de jeux Flash censé charger des jeux provenant de différents sites. Si le lecteur Flash n'intégrait pas de modèle de sécurité, il serait possible d'ajouter du code aux différents jeux chargés et de pirater ces derniers.

Si toutefois, nous devons accéder au code d'un jeu chargé, nous devons ajouter au sein de celui-ci une ligne de code autorisant le domaine spécifique à scripter l'animation. En d'autres termes, le contenu chargé doit autoriser le chargeur à le scripter.

Dans le cas de chargement d'images provenant d'autres domaines, celles-ci n'ont pas la possibilité d'autoriser le chargeur par ActionScript. Nous utilisons dans cas des fichiers XML contenant la liste des domaines autorisés. Ces fichiers sont appelés *fichiers de régulation*.

A retenir

- Par défaut, le lecteur Flash empêche de scripter ou modifier tout contenu graphique provenant d'un domaine différent.
- En revanche, la lecture seule d'animation ou images est toujours autorisée quelque soit le contexte.
- Le chargement de données type XML ou autres est toujours interdit dans un contexte interdomaine.

Programmation croisée

Nous entendons par le terme de *programmation croisée*, l'accès par ActionScript au code contenu par un SWF.

Nous avons vu au cours de la précédente partie comment faire communiquer deux animations situées sur un même domaine. Nous savons que par défaut, la communication est toujours autorisée entre deux SWF résidant sur le même domaine.

A l'inverse, lorsque deux animations souhaitent communiquer mais ne résident par sur le même domaine, l'animation devant être scriptée doit autoriser l'animation ayant amorcé le chargement.

La figure 13-13 illustre la situation :

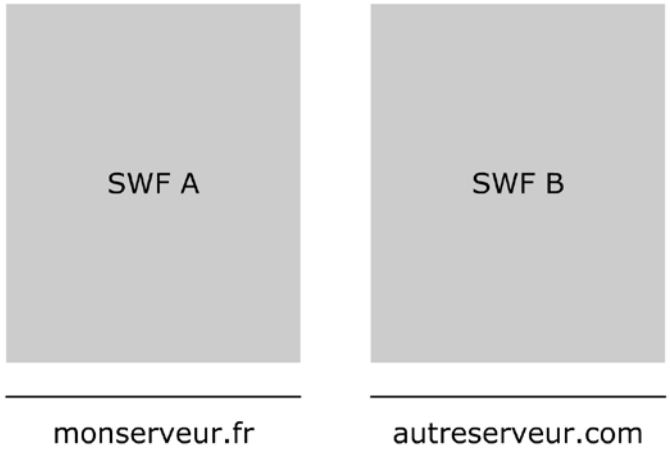

Figure 13-13. Animations en contexte interdomaine.

Dans le schéma illustré par la figure 13-15 nous voyons deux animations en contexte *interdomaine*.

L'animation `SWF A` souhaite scripter l'animation `SWF B`. Pour cela, cette dernière doit appeler la méthode `allowDomain` de la classe `flash.system.Security` dont voici la signature :

```
public static function allowDomain(... domains):void
```

Nous pouvons passer en paramètre les domaines autorisés à scripter le SWF en cours. Ainsi nous placerons au sein de l'animation `SWF B` le code suivant :

```
Security.allowDomain("monserveur.fr");
```

Il n'est pas nécessaire d'ajouter http devant le domaine, nous ne spécifions généralement que le nom et le domaine. Il est aussi possible de spécifier une adresse IP.

A retenir

- Dans un contexte interdomaine, l'accès par ActionScript au code contenu par un SWF est appelé *programmation croisée*.

Utiliser un fichier de régulation

Comme expliqué précédemment, dans le cas de chargement d'images bitmap provenant d'un serveur distant, il est impossible d'ajouter au sein de celles-ci un appel à la méthode `allowDomain`. Afin de pallier à ce problème, nous pouvons créer des fichiers de régulation.

Ces derniers doivent être placés sur le serveur hébergeant les images et sont en réalité de simples fichiers XML sauvés sous le nom de `crossdomain.xml`.

La figure 13-14 illustre l'idée :

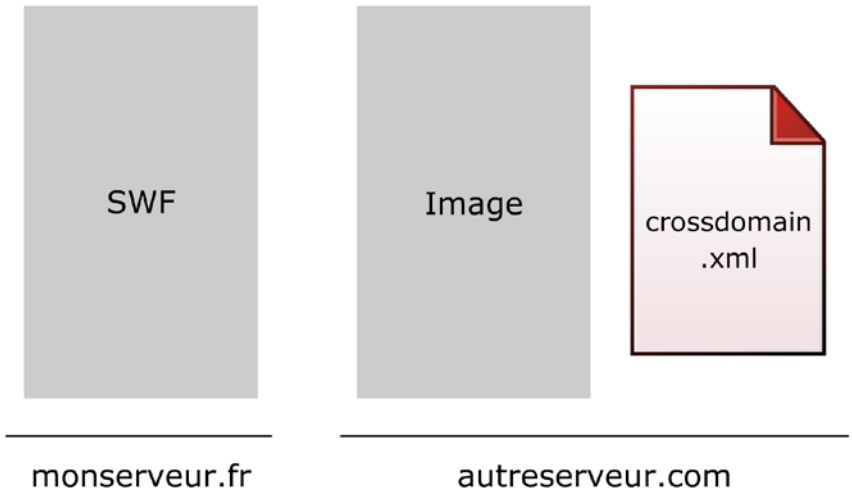

Figure 13-14. Fichier de régulation.

Lorsque le lecteur Flash charge une image, celui-ci tentait dans ses précédentes versions de charger automatiquement le fichier de régulation depuis la racine du serveur.

Depuis le lecteur 9, il est nécessaire d'indiquer s'il est nécessaire de charger le fichier de régulation avant de commencer le chargement de l'élément. L'utilisation de fichiers de régulation est fondamentale pour les sites hébergeant des images utilisées au sein d'applications Flash.

Voici une liste de sites utilisant un fichier de régulation :

- http://www.facebook.com/crossdomain.xml
- http://www.adobe.com/crossdomain.xml
- http://www.youtube.com/crossdomain.xml
- http://static.flickr.com/crossdomain.xml

En analysant le contenu d'un fichier de régulation nous découvrons un simple fichier XML contenant la liste des domaines autorisés à modifier les images. Ces derniers sont appelés plus couramment *domaines de confiance*.

Voici le contenu du fichier de régulation situé sur le serveur *YouTube* :

```
<cross-domain-policy>
<allow-access-from domain="*.youtube.com"/>
<allow-access-from domain="*.google.com"/>
</cross-domain-policy>
```

Ce fichier de régulation indique que seuls les SWF hébergés dans des sous-domaines de youtube.com ou google.com peuvent scripter les images hébergées sur youtube.com.

En analysant le fichier de régulation de flickr, nous remarquons que ces derniers sont beaucoup plus permissifs :

```
<cross-domain-policy>
<allow-access-from domain="*"/>
</cross-domain-policy>
```

Nous découvrons que tous les domaines peuvent modifier les images hébergées sur flickr.com.

A retenir

- Dans le cas de chargement d'images distantes, des fichiers de régulation peuvent être créés afin d'autoriser les domaines de confiance.
- Un fichier de régulation est un simple fichier XML.
- Celui-ci doit être nommé par défaut `crossdomain.xml`.

Contexte de chargement

Lorsque les méthodes `load` et `loadBytes` de la classe `Loader` sont appelées, nous pouvons passer en deuxième paramètre un contexte de chargement exprimé par la classe `flash.system.LoaderContext`.

La classe `LoaderContext` permet d'indiquer le domaine d'application et de sécurité dans lequel sera placé le contenu.

Le constructeur de la classe `LoaderContext` accepte trois paramètres dont voici le détail :

- `checkPolicyFile` : indique si un fichier de régulation doit être chargé avant de commencer à charger le contenu.
- `applicationDomain` : sert à préciser le domaine d'application à utiliser une fois le contenu chargé. La notion de domaine d'application est traitée dans la partie intitulée *Bibliothèque partagée*.
- `securityDomain` : représente le modèle de sécurité. Il est seulement utilisé lors du chargement de fichiers SWF. Son rôle est traité dans la partie intitulée *Bibliothèque partagée*.

Ainsi, afin de pouvoir modifier une image hébergée depuis un domaine distant nous demandons au lecteur Flash de charger un fichier de régulation :

```
var chargeur:Loader = new Loader();

var requete:URLRequest = new URLRequest
("http://www.serveurdistant.com/images/wiiflash.jpg");

var contexte:LoaderContext = new LoaderContext ( true );

chargeur.load ( requete, contexte );
```

Automatiquement, le lecteur Flash tente de charger un fichier de régulation stocké à la racine du serveur `serveurdistant`. Si un tel fichier nommé `crossdomain.xml` est présent et autorise notre domaine alors nous pouvons modifier l'image chargée.

Si aucun fichier de régulation n'est trouvé, le lecteur Flash empêche toute modification de l'image chargée.

Pour des questions de pratique, il vous est peut-être impossible de placer un fichier de régulation à la racine du domaine distant. Si le

fichier de régulation est placé à un emplacement différent de la racine du domaine distant. Nous spécifions son chemin à l'aide de la méthode `loadPolicyFile` de la classe `Security`.

Dans le code suivant, nous spécifions au lecteur de ne pas chercher le fichier de régulation à la racine du domaine mais au sein du répertoire `images` :

```
var chargeur:Loader = new Loader();

Security.loadPolicyFile("http://serveurdistant.com/images/regulation.xml");

var requete:URLRequest = new URLRequest
("http://www.serveurdistant.com/images/wiiflash.jpg");

var contexte:LoaderContext = new LoaderContext ( true );

chargeur.load ( requete, contexte );
```

Notons que grâce à la méthode `loadPolicyFile` nous pouvons aussi spécifier le nom du fichier de régulation, ici `regulation.xml`.

Attention, l'emplacement du fichier de régulation a une importance. A la racine, celui-ci autorise l'accès à tous les fichiers du site. S'il se trouve plus loin dans l'arborescence, il n'autorisera que les fichiers de ce dossier ainsi que ceux des dossiers enfants.

Contourner les restrictions de sécurité

Nous avons vu dans la partie intitulée *Programmation croisée* qu'il était possible d'autoriser la manipulation d'images hébergées sur des domaines distants par la création de fichiers de régulation.

Malheureusement, dans certains cas, l'ajout de tels fichiers est impossible. Certains sites prévoient quelquefois leur installation, mais ils demeurent minoritaires. Il peut donc être intéressant de savoir contourner les restrictions de sécurité du lecteur Flash. C'est ce que nous allons appendre dès maintenant.

Imaginons le cas suivant :

Vous venez d'apprendre que vous devez développer une application Flash basée sur des images provenant de différentes sources. En d'autres termes, chaque image devra être chargée tout en étant hébergée sur n'importe quel domaine.

Pour l'instant, cela ne pose aucun problème car le simple chargement d'images en situation *interdomaine* est autorisé. En cours de développement vous vous rendez compte qu'un lissage est nécessaire et qu'il serait intéressant de pouvoir l'activer auprès des images chargées.

Un premier réflexe vous incite à activer la propriété `smoothing` sur l'objet `Bitmap` chargé :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine );

// image google map
var requete:URLRequest = new URLRequest (
"http://kh0.google.fr/kh?n=404&v=22&t=trtqttqqrrqrqssts" );

// chargement de l'image
chargeur.load ( requete );

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

function termine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // accès à l'image bitmap
  var image:Bitmap = Bitmap ( objetLoaderInfo.content );

  // activation du lissage
  image.smoothing = true;

}
```

En testant l'application en local, le lissage fonctionne, la figure 13-15 illustre le résultat :

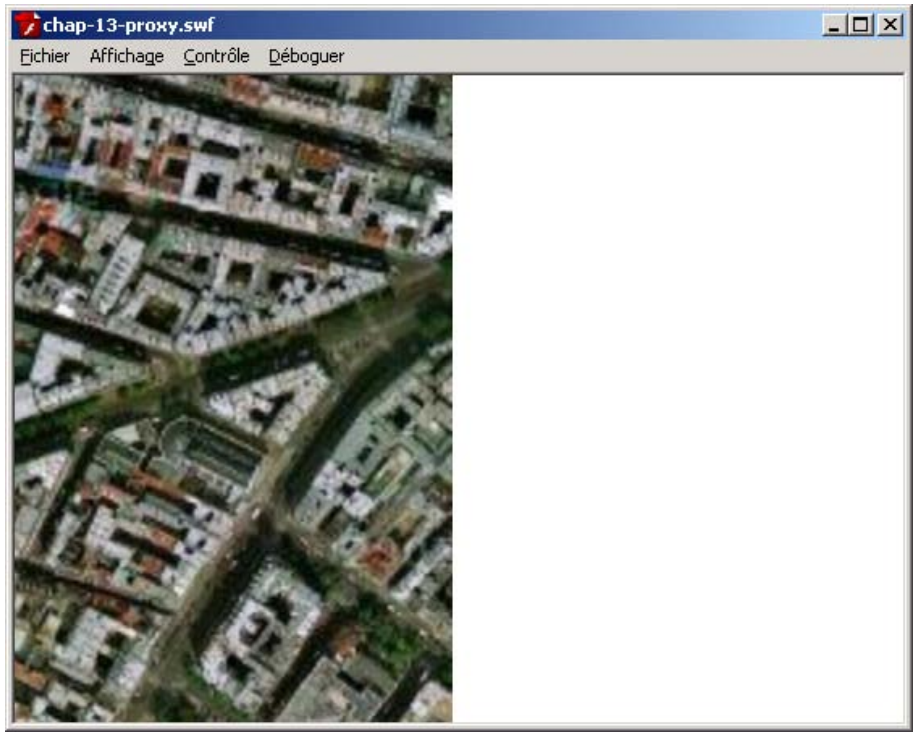

Figure 13-15. Image lissée provenant de Google Map.

Malheureusement, une fois l'application publiée sur votre serveur tout accès au contenu chargé lève une erreur à l'exécution de type `SecurityError` :

```
SecurityError: Error #2122: Violation de la sécurité Sandbox :
LoaderInfo.content : http://www.bytearray.org/pratique-as3/chap-13-proxy.swf
ne peut pas accéder à http://kh0.google.fr/kh?n=404&v=22&t=trtqttqqrrqrqssts.
Un fichier de régulation est nécessaire, mais l'indicateur checkPolicyFile
n'a pas été défini lors du chargement de ce support.
```

Ceci est dû au fait que le serveur Google ne possède aucun fichier de régulation nous autorisant à manipuler ses images. L'utilisation de la propriété `checkPolicyFile` est donc dans ce cas précis inutile. En d'autres termes, le contenu chargé est en lecture seule.

Nous allons donc utiliser une astuce consistant à faire croire au lecteur Flash que nous chargeons un élément provenant du même domaine. Pour cela, nous utilisons un relais, plus couramment appelé *serveur mandataire*.

La figure 13-16 illustre le concept :

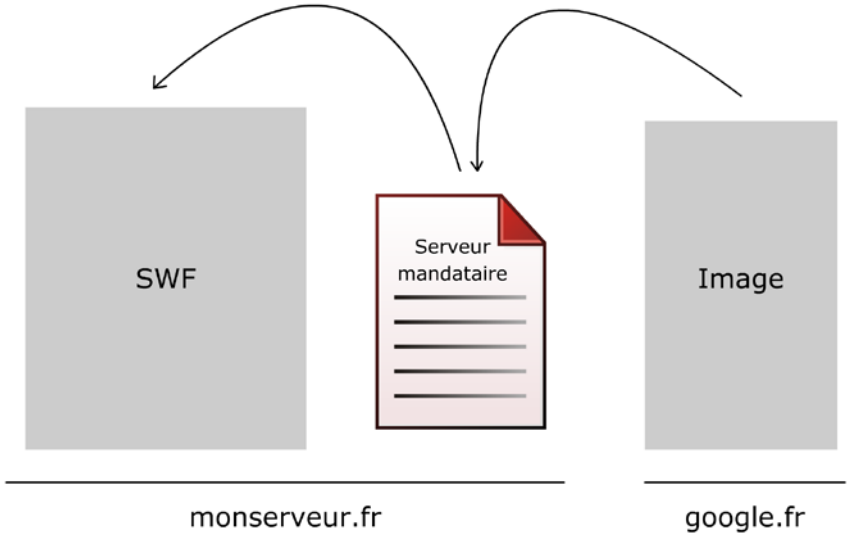

Figure 13-16. Chargement d'images par serveur mandataire.

L'astuce consiste à charger l'image depuis le serveur mandataire, qui est ensuite chargé par le lecteur Flash. Ce dernier pense charger un élément provenant du même domaine, sans penser que le serveur mandataire contient l'image provenant du domaine distant.

La création du serveur mandataire se limite à deux lignes de code PHP. Nous utilisons dans notre exemple le langage serveur PHP qui s'avère être un des langages les plus efficace pour travailler avec Flash.

Au sein d'un fichier intitulé `proxy.php` nous ajoutons le script suivant :

```
<?php

 $chemin = $_POST["chemin"];
 readfile($chemin);

?>
```

Nous passons par le tableau `$_POST` le chemin d'image. Puis la fonction PHP `readfile` renvoie le flux d'image directement au lecteur Flash.

Puis nous modifions le code, afin de charger l'image par le serveur mandataire :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE, termine);
```

```
// image google map
var requete:URLRequest = new URLRequest ( "proxy.php" );

// création d'un objet URLVariables permettant
// de passer des variables au serveur mandataire
var variables:URLVariables = new URLVariables();

// création de la variable chemin
variables.chemin = "http://kh0.google.fr/kh?n=404&v=22&t=trtqttqqrrqrqssts";

// les variables doivent être passées par la requete HTTP
requete.data = variables;

// les variables sont envoyées au sein du tableau POST
requete.method = URLRequestMethod.POST;

// chargement de l'image
chargeur.load ( requete );

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

function termine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // accès à l'image bitmap chargée
  var image:Bitmap = Bitmap ( objetLoaderInfo.content );

}
```

Nous utilisons la classe `URLVariables` afin de passer l'adresse de l'image à charger au serveur mandataire. Nous reviendrons en détail sur cette classe au cours du chapitre 14 intitulé *Chargement et envoi de données*.

Une fois publiée, si nous lançons l'application, celle-ci ne lève plus d'erreur à l'exécution lorsque nous accédons à la propriété `content`.

Nous pouvons ainsi activer le lissage en passant la valeur booléenne `true` à la propriété `smoothing` de l'objet `Bitmap` chargé :

```
function termine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // accès à l'image bitmap chargée
  var image:Bitmap = Bitmap ( objetLoaderInfo.content );

  // activation du lissage
  image.smoothing = true;

}
```

La figure 13-17 illustre la différence entre l'image lissée et non lissée :

Figure 13-17. Image non lissée et lissée.

De la même manière, il peut être nécessaire de rendre sous forme bitmap un objet `Loader` contenant une image provenant d'un domaine distant :

```
function termine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // création d'une instance de BitmapData
  var donneesBitmap:BitmapData = new BitmapData ( objetLoaderInfo.width,
objetLoaderInfo.height );

  // l'objet Loader est rendu sous forme bitmap
  donneesBitmap.draw ( pEvt.target.loader );

  var image:Bitmap = new Bitmap ( donneesBitmap );

  image.x = objetLoaderInfo.width + 5;

  addChild ( image );

}
```

Ce qui génère le résultat illustré en figure 13-18 :

Figure 13-18. Image dupliquée.

Sans serveur mandataire, l'appel de la méthode `draw` aurait levé une erreur de sécurité à l'exécution.

Cette technique de serveur mandataire est une solution efficace qui possède malheureusement un inconvénient. Au lieu d'être directement chargée depuis le client, l'image est d'abord chargée par le serveur puis chargée par le client. La charge serveur peut donc être plus importante et à surveiller sur un grand projet destiné à un trafic important.

A retenir

- L'utilisation d'un serveur mandataire permet de charger et modifier tout type de contenu provenant d'un différent domaine.
- C'est une solution simple et efficace mais qui peut entraîner une charge serveur plus importante.

Bibliothèque partagée

Dans le cas de chargement d'animations, il peut être parfois utile d'extraire une classe utilisée au sein d'un SWF afin de l'utiliser au sein de l'animation procédant au chargement. Cela est rendu possible grâce au mécanisme de *bibliothèque partagée à l'exécution* apporté par la classe `flash.system.ApplicationDomain`.

Dans un nouveau document Flash nous créons un symbole clip comme illustré en figure 13-19 :

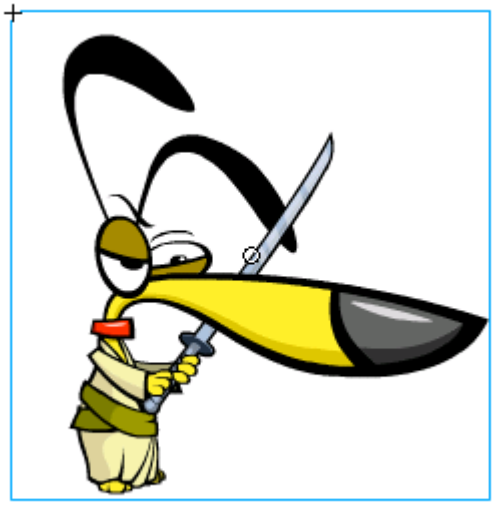

Figure 13-19. Symbole clip.

Puis nous lions le symbole à une classe `Ninja` grâce au panneau de *Propriétés de liaisons* :

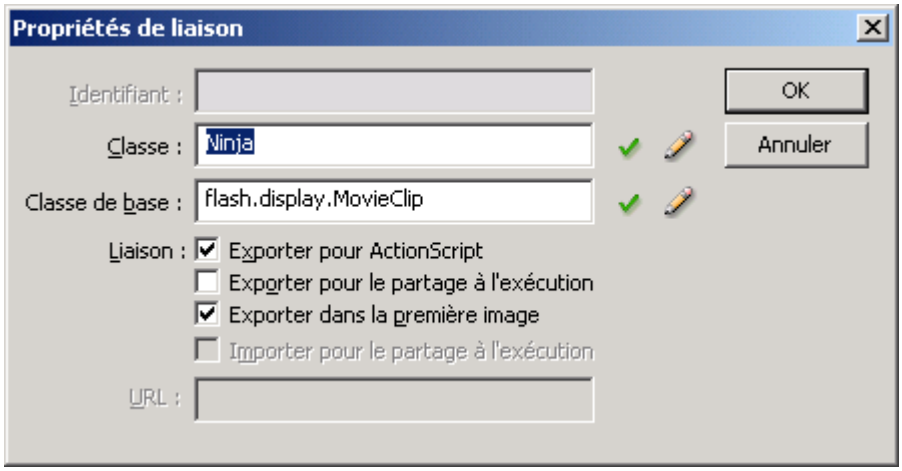

Figure 13-20. Panneau de propriétés de liaison.

Une fois définie, nous exportons simplement l'animation sous le nom de `bibliotheque.swf`. Nous allons maintenant charger ce fichier SWF et accéder dynamiquement à la classe `Ninja`.

Dans un nouveau document Flash, nous créons un objet `Loader` puis nous chargeons l'animation `bibliotheque.swf` :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
```

```
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

// chargement de l'animation contenant la classe Ninja
chargeur.load ( new URLRequest ("bibliotheque.swf") );

function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // affiche : [object LoaderInfo]
  trace( objetLoaderInfo );

}
```

Une fois l'animation chargée, nous pouvons accéder à toutes les classes définies au sein de celle-ci grâce à la méthode `getDefinition` de la classe. Celle-ci peut être appelée dès lors que l'événement `Event.INIT` est diffusé. Nous n'ajoutons pas volontairement l'objet `Loader` à a liste d'affichage car nous souhaitons simplement extraire une classe partagée.

Souvenez-vous, nous avons vu précédemment que la classe `LoaderInfo` possède une propriété `applicationDomain` utilisée lors du chargement de SWF.

Celle-ci référence un objet appelé *Domaine d'application* :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // référence le domaine d'application du SWF chargé
  var domaineApplication:ApplicationDomain =
objetLoaderInfo.applicationDomain;

  // affiche : [object ApplicationDomain]
  trace( domaineApplication );

}
```

Le domaine d'application est un objet dans lequel sont placés toutes les définitions de classe d'un SWF. Ainsi, le domaine d'application de l'animation `bibliotheque.swf` contient une classe `Ninja`.

La classe `ApplicationDomain` possède deux méthodes dont voici le détail :

- `getDefinition` : Extrait une définition de classe spécifique.
- `hasDefinition` : Indique si la définition de classe existe.

Deux propriétés peuvent aussi être utilisées :

- `currentDomain` : Référence le domaine d'application du SWF en cours.
- `parentDomain` : Référence le domaine d'application parent.

Nous allons extraire la classe `Ninja` du domaine d'application du SWF chargé puis l'instancier et afficher le symbole au sein de l'animation procédant au chargement :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // référence le domaine d'application du SWF chargé
  var domaineApplication:ApplicationDomain =
objetLoaderInfo.applicationDomain;

  // extrait la définition de classe Ninja
  var importNinja:Class = Class ( domaineApplication.getDefinition( "Ninja" )
);

  // création d'une instance de Ninja
  var instanceNinja:DisplayObject = new importNinja();

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

}
```

La figure 13-21 illustre le résultat :

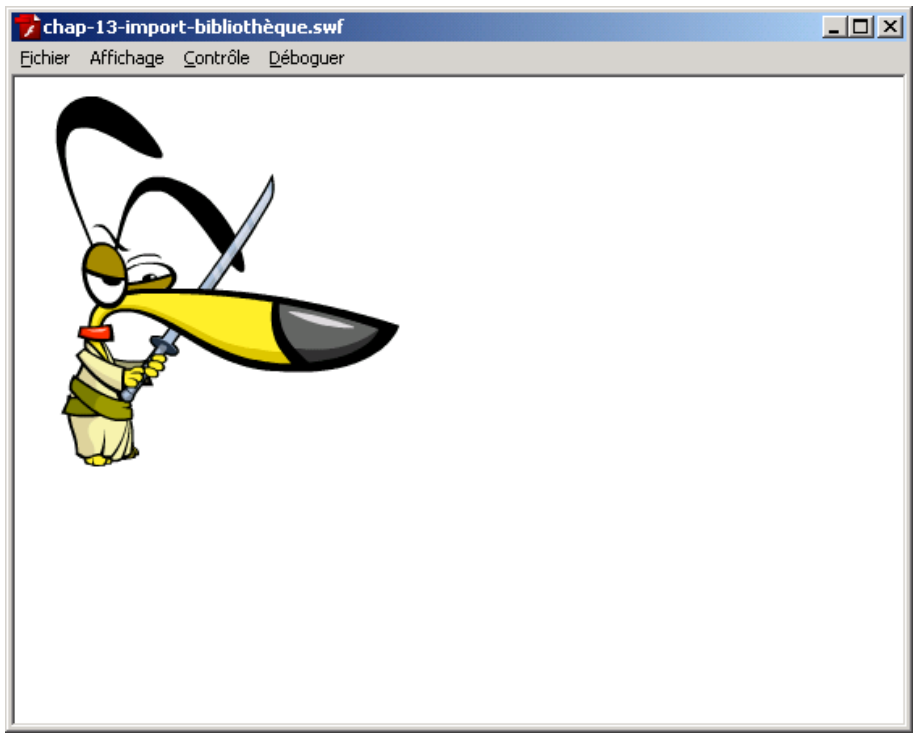

Figure 13-21. Affichage du symbole `Ninja`.

Si nous tentons d'extraire une classe inexistante :

```
// tente d'extraire une définition classe nommée Nina
var importNinja:Class = Class ( domaineApplication.getDefinition( "Nina" ) );
```

Une erreur de type `ReferenceError` est levée :

```
ReferenceError: Error #1065: La variable Nina n'est pas définie.
```

A l'aide d'un bloc, `try catch` nous pouvons gérer l'erreur et ainsi réagir :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // référence le domaine d'application du SWF chargé
  var domaineApplication:ApplicationDomain =
objetLoaderInfo.applicationDomain;

  try

  {

        // tentative d' extraction de la définition de classe
        var importNinja:Class = Class ( domaineApplication.getDefinition(
"Nina" ) );

        // création d'une instance de Ninja
        var instanceNinja:DisplayObject = new importNinja();

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

  } catch ( pError:Error )

  {

        trace("La définition de classe spécifiée n'est pas disponible");

  }

}
```

Si l'utilisation d'un bloc `try catch` ne vous convient pas, l'appel de la méthode `hasDefinition` offre un résultat équivalent :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

// chargement de l'animation contenant la classe Ninja
chargeur.load ( new URLRequest ("librairie.swf") );
```

```
var definitionClasse:String = "Ninja";

function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // référence le domaine d'application du SWF chargé
  var domaineApplication:ApplicationDomain =
objetLoaderInfo.applicationDomain;

  // vérifie si la définition de classe Ninja est disponible
  if ( domaineApplication.hasDefinition( definitionClasse ) )

  {

        // tentative d' extraction de la définition de classe
        var importNinja:Class = Class ( domaineApplication.getDefinition(
definitionClasse ) );

        // création d'une instance de Ninja
        var instanceNinja:DisplayObject = new importNinja();

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

  } else trace ("La définition de classe " +  definitionClasse + " n'est pas
disponible");

}
```

L'extraction de classes est rendue possible car l'animation contenant les classes à extraire provient du même domaine que l'animation procédant au chargement.

Bien entendu, le modèle de sécurité du lecteur Flash empêche par défaut l'extraction de classes entre deux SWF évoluant dans un contexte interdomaine. Dans ce cas, nous devons explicitement demander au lecteur Flash de placer le SWF chargé dans le même domaine de sécurité afin de pouvoir extraire les classes.

Une première approche consiste à appeler la méthode `allowDomain` de la classe `Security` depuis le SWF dont les classes sont extraites.

```
Security.allowDomain("monserveurDeConfiance.fr");
```

La seconde requiert le placement d'un fichier de régulation sur le domaine du SWF à charger, puis de passer un objet `LoaderContext` à la méthode `load` de l'objet `Loader` en spécifiant le domaine de sécurité en cours par la propriété `securityDomain` :

```
// création de l'objet Loader
var chargeur:Loader = new Loader();

// écoute de la fin du chargement
chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );
```

```
// création d'un objet de contexte
var contexte:LoaderContext = new LoaderContext();

// nous demandons de placer le SWF chargé dans le même domaine
// de sécurité afin de pouvoir extraire ses classes
contexte.securityDomain = SecurityDomain.currentDomain;

// chargement de l'animation contenant la classe Ninja
// en spécifiant le contexte
chargeur.load ( new URLRequest
("http://serveurdistant.com/swf/bibliotheque.swf"), contexte );

function chargementTermine ( pEvt:Event ):void

{


  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // référence le domaine d'application du SWF chargé
  var domaineApplication:ApplicationDomain =
objetLoaderInfo.applicationDomain;

  // extrait la définition de classe Ninja
  var importNinja:Class = Class ( domaineApplication.getDefinition( "Ninja" )
);

  // création d'une instance de Ninja
  var instanceNinja:DisplayObject = new importNinja();

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

}
```

Si pour des questions de pratique, vous n'avez pas la possibilité d'appeler la méthode `allowDomain` de la classe `Security` ou de placer un fichier de régulation sur le domaine distant, l'utilisation d'un fichier serveur mandataire est ici aussi envisageable.

Grâce à ce concept d'import dynamique de classes, nous pouvons imaginer toutes sortes d'applications tirant profit d'une telle fonctionnalité. Une application Flash pourrait importer, dès son initialisation un SWF contenant les définitions de classe nécessaires. Celles-ci seraient dynamiquement instanciées puis utilisées dans l'application.

L'application reposerait donc entièrement sur ces classes importées dynamiquement. Afin de mettre à jour l'application, nous pourrions simplement régénérer le SWF contenant les définitions de classe.

L'application pourrait être mise à jour de la même manière que des `.dll` dans d'autres langages comme C++ ou C#.

A retenir

- La méthode `getDefinition` de la classe `ApplicationDomain` permet d'extraire une définition de classe contenue dans un SWF.
- Cette extraction est soumise au modèle de sécurité du lecteur Flash.
- Afin de pouvoir extraire une classe d'un SWF distant, celui-ci doit autoriser l'animation ayant amorcé le chargement par la méthode `allowDomain` de la classe `Security` ou la création d'un fichier de régulation.

Désactiver une animation chargée

Très souvent, un site Flash est constitué d'une application principale chargeant différents modules, séparés en plusieurs SWF. Chacun d'entre eux est ensuite chargé afin de naviguer dans le site.

Le fonctionnement de la classe `Loader` nous réserve encore quelques surprises. En réalité, la méthode `unload` vide le contenu chargé mais ne le désactive pas. Cela diffère du traditionnel `loadMovie` utilisé en ActionScript 1 et 2, qui remplaçait le contenu précédemment chargé en désactivant tous les objets contenus.

En ActionScript 3, lorsque la méthode unload est exécutée, le contenu chargé est simplement supprimé de la liste d'affichage. La seule référence à l'animation est celle que possède l'objet `Loader`. Si nous supprimons son contenu il n'existe alors plus aucune référence vers l'animation. Celle-ci va donc demeurer et vivre en mémoire jusqu'à ce que le ramasse miettes intervienne et la supprime définitivement.

Ainsi, au chargement d'une nouvelle rubrique, le son de la précédente continuerait de jouer. De la même manière, tous les événements souscrits continueraient d'être diffusés.

Il faut donc prévoir *obligatoirement* un mécanisme de désactivation comme nous l'avons fait jusqu'à présent pour les objets d'affichage.

Afin de correctement désactiver une animation, nous utilisons l'événement `Event.UNLOAD` diffusé par l'objet `LoaderInfo` associé au SWF en cours. Le code suivant doit donc être placé au sein du SWF à désactiver :

```
// écoute la suppression de l'animation
loaderInfo.addEventListener ( Event.UNLOAD, desactivation );

function desactivation ( pEvt:Event ):void

{

  // logique de désactivation de l'animation
  // désactivation des événements, du son, objets videos etc
```

```
}
```

Lorsque la méthode `unload` est exécutée, ou qu'une nouvelle animation est chargée, l'événement `Event.UNLOAD` est diffusé au sein de l'animation chargée. Nous intégrons au sein de la fonction écouteur `desactivation` la logique nécessaire afin de désactiver totalement l'animation en cours.

Dans le code suivant, nous stoppons le son en cours de lecture :

```
// création d'un objet Sound
var monSon:Sound = new Sound();

// chargement du son
monSon.load ( new URLRequest ("son.mp3") );

// création d'un objet SoundChannel par l'appel de la méthode Sound.play()
var canalAudio:SoundChannel = monSon.play();

// écoute la suppression de l'animation
loaderInfo.addEventListener ( Event.UNLOAD, desactivation );

function desactivation ( pEvt:Event ):void

{

  // logique de désactivation de l'animation
  // désactivation des événements, du son, objets videos etc
  canalAudio.stop();

}
```

L'animation chargée étant supprimée de la liste d'affichage, l'écoute de l'événement `Event.REMOVED_FROM_STAGE` est aussi envisageable :

```
// création d'un objet Sound
var monSon:Sound = new Sound ();

// chargement du son
monSon.load ( new URLRequest ("son.mp3") );

// création d'un objet SoundChannel par l'appel de la méthode Sound.play()
var canalAudio:SoundChannel = monSon.play();

// écoute la suppression de l'animation
addEventListener ( Event.REMOVED_FROM_STAGE, desactivation );

function desactivation ( pEvt:Event ):void

{

  // logique de désactivation de l'animation
  // désactivation des événements, du son, objets videos etc
  canalAudio.stop();

}
```

Ce comportement peut poser de graves problèmes lorsque vous n'êtes pas l'auteur du contenu chargé. Vous êtes donc dans l'incapacité d'intégrer un mécanisme de désactivation. Il n'existe aujourd'hui aucune solution viable permettant de désactiver automatiquement un contenu tiers.

A retenir

- Lorsque la méthode `unload` est appelée, le contenu est supprimé de la liste d'affichage mais n'est pas désactivé.
- Il est impératif de prévoir un mécanisme de désactivation au sein des animations chargées.
- Il n'est pas possible de désactiver automatiquement un contenu tiers.

Communication AVM1 et AVM2

La machine virtuelle ActionScript 3 (AVM2) offre la possibilité de lire des animations développées en ActionScript 1 et 2 (AVM1). Celles-ci sont alors considérées comme des objets de type `flash.display.AVM1Movie`.

Dans le cas d'un portail de jeux vidéo développé en ActionScript 3, la majorité des jeux chargés seront d'ancienne génération, développés en ActionScript 1 ou 2. Malheureusement, l'échange entre les deux animations n'est pas simplifié.

Si nous tentons d'accéder au contenu de ces derniers, le lecteur lève une erreur indiquant que l'accès est impossible. Il faut considérer un objet `AVM1Movie` comme un objet hermétique ne pouvant être pénétré.

Afin de mettre en évidence ce comportement, nous allons créer une animation ActionScript 1 ou 2 et y intégrer une simple animation. L'animation est représentée par une instance du symbole `Garcon` utilisé lors du chapitre précédent. Nous lui donnons `animation` comme nom d'occurrence.

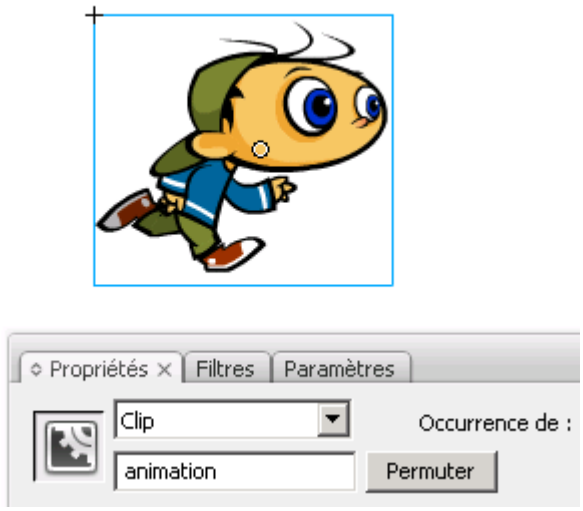

Figure 13-22. Instance du symbole `Garcon`.

Nous allons depuis l'animation ActionScript 3, communiquer avec le contenu l'animation d'ancienne génération pour stopper l'animation du clip `animation`.

Dans le code suivant, nous chargeons l'animation d'ancienne génération :

```
var chargeur:Loader = new Loader();

chargeur.contentLoaderInfo.addEventListener ( Event.COMPLETE,
chargementTermine );

chargeur.load ( new URLRequest ("anim-vm1.swf" ));

addChild ( chargeur );

function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );

  // affiche : [object AVM1Movie]
  trace( objetLoaderInfo.content );

}
```

Nous remarquons que la propriété `content` de l'objet `LoaderInfo` nous renvoie un objet de type `AVM1Movie`. Si nous tentons de pénétrer à l'intérieur de l'animation :

```
function chargementTermine ( pEvt:Event ):void

{

  var objetLoaderInfo:LoaderInfo = LoaderInfo ( pEvt.target );
```

```
        var contenu:DisplayObject = objetLoaderInfo.content;

        if ( contenu is AVM1Movie )

        {

              var animationVM1:AVM1Movie = AVM1Movie ( contenu );

              // tentative d'accès à l'occurrence animation
              trace( animationVM1.animation );

        }

      }
```

L'erreur à la compilation suivante est générée :

```
1119: Accès à la propriété animation peut-être non définie, via la référence
de type static flash.display:AVM1Movie.
```

Afin d'accéder au contenu de l'animation chargée, nous devons passer par un moyen détourné. Deux classes vont nous permettre de communiquer :

- `flash.net.LocalConnection` : la classe `LocalConnection` permet d'échanger des données entre différents SWF distincts.
- `flash.external.ExternalInterface` : la classe `ExternalInterface` permet la communication entre le code ActionScript et la page contenant le lecteur Flash.

Nous allons utiliser pour cet exemple la classe `LocalConnection` qui s'avère être la solution la plus souple, en ne nécessitant pas de code JavaScript contrairement à la classe `ExternalInterface`.

Nous commençons par créer une instance de la classe `LocalConnection` dans l'animation dans laquelle nous souhaitons accéder :

```
// création de l'objet récepteur
var recepteur:LocalConnection = new LocalConnection();

// connexion au canal utilisé par l'émetteur
recepteur.connect ("canalCommunication");

// définition de la méthode appelée par l'émetteur
recepteur.stopAnimation = function ( )

{

  animation.stop();

}
```

Puis au sein de l'animation souhaitant initier la communication, nous ajoutons une nouvelle instance de la classe `LocalConnection` afin d'émettre les messages :

```
var chargeur:Loader = new Loader();

chargeur.load ( new URLRequest ("anim-vm1.swf" ));

addChild ( chargeur );

// création de l'objet émetteur
var emetteur:LocalConnection = new LocalConnection();

boutonStop.addEventListener ( MouseEvent.CLICK, clicBouton );

function clicBouton ( pEvt:MouseEvent ):void

{

  // émission d'un message pour exécuter la méthode stopAnimation par le
canal canalCommunication
  emetteur.send ("canalCommunication ", "stopAnimation");

}
```

Lorsque nous cliquons sur le bouton `boutonStop`, un message est envoyé à l'animation chargée par l'appel de la méthode `send` de l'objet `LocalConnection`.

A retenir

- La communication entre deux animations AVM1 et AVM2 est possible par l'intermédiaire de la classe `LocalConnection` ou `ExternalInterface`.

Nous savons désormais comment charger du contenu graphique au sein du lecteur Flash. Passons maintenant au chargement et à l'envoi de données en ActionScript 3.