10

Diffusion d'événements personnalisés

L'histoire

Dans une application ActionScript, la communication inter objets se réalise majoritairement par la diffusion d'événements natifs ou personnalisés. Mais qu'est ce qu'un événement personnalisé ?

Il s'agit d'un événement qui n'existe pas au sein du lecteur Flash mais que nous ajoutons afin de gérer les différentes interactions entre nos objets. Depuis très longtemps ActionScript intègre différentes classes permettant de diffuser nos propres événements.

La première fut `AsBroadcaster` intégrée depuis le lecteur Flash 5, son implémentation native l'avait rendu favorite des développeurs ActionScript. Sa remplaçante `BroadcasterMX` introduite plus tard par Flash MX et codée en ActionScript fut moins bien accueillie. Aujourd'hui ces classes n'existent plus en ActionScript 3 et sont remplacées par la classe native `flash.events.EventDispatcher`.

Thibault Imbert
pratiqueactionscript3.bytearray.org

Au sein d'une interface graphique nous pourrions imaginer un bouton diffusant en événement indiquant son état actif ou inactif. Une classe générant des fichiers PDF pourrait diffuser des événements relatifs au processus de création du fichier. Ainsi, la diffusion d'événements personnalisés constitue la suite logique à la création d'objets personnalisés.

La classe EventDispatcher

Comme nous le voyons depuis le début de l'ouvrage, ActionScript 3 est un langage basé sur un modèle événementiel appelé *Document Object Model*. Celui-ci repose sur la classe `EventDispatcher` dont toutes les classes natives de l'API du lecteur Flash héritent.

Dans le code suivant nous voyons la relation entre la classe `EventDispatcher` et deux classes graphiques :

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

// affiche : true
trace( monSprite is EventDispatcher );

var monClip:MovieClip = new MovieClip();

// affiche : true
trace( monClip is EventDispatcher );
```

Rappelez-vous, toutes les classes issues du paquetage `flash` sont des sous-classes d'`EventDispatcher` et possèdent donc ce type commun.

Nous créons un `Sprite` puis nous écoutons un événement personnalisé auprès de ce dernier :

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

// écoute de l'événement monEvent
monSprite.addEventListener ( "monEvenement", ecouteur );

// fonction écouteur
function ecouteur ( pEvt:Event ):void

{

  trace( pEvt );

}
```

La classe `Sprite` ne diffuse pas par défaut d'événement nommé `monEvenement`, mais nous pouvons le diffuser simplement grâce à la méthode `dispatchEvent` dont voici la signature :

```
public function dispatchEvent(event:Event):Boolean
```

En ActionScript 2 la méthode `dispatchEvent` diffusait un objet événementiel non typé. Nous utilisions généralement un objet littéral auquel nous ajoutions différentes propriétés manuellement.

En ActionScript 3, afin de diffuser un événement nous devons créer un objet événementiel, représenté par une instance de la classe `flash.events.Event` :

```
// création de l'objet événementiel
var objetEvenementiel:Event = new Event (type, bubbles, cancelable);
```

Le constructeur de la classe `Event` accepte trois paramètres :

- `type` : le nom de l'événement à diffuser.
- `bubbles` : indique si l'événement participe à la phase de remontée.
- `cancelable` : indique si l'événement peut être annulé.

Dans la plupart des situations nous n'utilisons que le premier paramètre `type` de la classe `Event`.

Une fois l'objet événementiel créé, nous le passons à la méthode `dispatchEvent` :

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

// écoute de l'événement monEvent
monSprite.addEventListener ( "monEvenement", ecouteur );

// fonction écouteur
function ecouteur (pEvt:Event):void

{

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

}

// création de l'objet événementiel
var objetEvenementiel:Event = new Event ("monEvenement", bubbles,
cancelable);

// nous diffusons l'événement monEvenement
monSprite.dispatchEvent (objetEvenementiel);
```

Généralement, nous ne créons pas l'objet événementiel séparément, nous l'instancions directement en paramètre de la méthode `dispatchEvent` :

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

// écoute de l'événement monEvent
monSprite.addEventListener ("monEvenement", ecouteur );
```

```
// fonction écouteur
function ecouteur ( pEvt:Event ):void

{

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

}

// nous diffusons l'événement monEvenement
monSprite.dispatchEvent ( new Event ("monEvenement") );
```

Cet exemple nous montre la facilité avec laquelle nous pouvons diffuser un événement en ActionScript 3.

A retenir

- Le modèle événementiel ActionScript 3 repose sur la classe `EventDispatcher`.
- Toutes les classes issues du paquetage `flash` peuvent diffuser des événements natifs ou personnalisés.
- Afin de diffuser un événement, nous utilisons la méthode `dispatchEvent`.
- La méthode `dispatchEvent` accepte comme paramètre une instance de la classe `Event`.

Mise en application

Nous allons développer une classe permettant à un symbole de se déplacer dans différentes directions. Lorsque celui-ci arrive à destination nous souhaitons diffuser un événement approprié.

A côté d'un nouveau document Flash CS3, nous sauvons une classe nommée `Balle.as` contenant le code suivant :

```
package

{

  import flash.display.Sprite;

  // la classe Balle étend la classe Sprite
  public class Balle extends Sprite

  {

       public function Balle ()

       {

            trace( this );
```

```
        }

    }

}
```

Nous lions notre symbole de forme circulaire à celle-ci par le panneau *Propriétés de liaison*. Puis nous instancions le symbole :

```
// création du symbole
// affiche : [object Balle]
var maBalle:Balle = new Balle();

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

A chaque clic souris sur la scène, la balle doit se diriger vers le point cliqué. Nous devons donc écouter l'événement `MouseEvent.CLICK` de manière globale auprès de l'objet `Stage`.

Nous avons vu au cours du précédent chapitre comment accéder de manière sécurisée à l'objet `Stage`. Nous intégrons le même mécanisme dans la classe `Balle` :

```
package

{

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

  // la classe Balle étend la classe Sprite
  public class Balle extends Sprite

  {

        public function Balle ()

        {
              // écoute de l'événement Event.ADDED_TO_STAGE
              addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );

        }

        private function ajoutAffichage( pEvt:Event ):void

        {

              // écoute de l'événement MouseEvent.CLICK
              stage.addEventListener ( MouseEvent.CLICK, clicSouris );

        }

        private function clicSouris ( pEvt:MouseEvent ):void

        {
```

```
            // affiche : [MouseEvent type="click" bubbles=true
cancelable=false eventPhase=2 localX=81 localY=127 stageX=81 stageY=127
relatedObject=null ctrlKey=false altKey=false shiftKey=false delta=0]
            trace( pEvt );

      }

  }

}
```

A chaque clic sur la scène, l'événement `MouseEvent.CLICK` est diffusé, la fonction écouteur `clicSouris` est alors déclenchée.

Nous allons intégrer à présent la notion de mouvement. Nous définissons deux propriétés `sourisX` et `sourisY` au sein de la classe. Celles-ci vont nous permettre de stocker la position de la souris :

```
// stocke les coordonnées de la souris
private var sourisX:Number;
private var sourisY:Number;
```

Puis nous modifions la méthode `clicSouris` afin d'affecter ces propriétés :

```
package

{

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

  // la classe Balle étend la classe Sprite
  public class Balle extends Sprite

  {

      // stocke les coordonnées de la souris
      private var sourisX:Number;
      private var sourisY:Number;

      public function Balle ()

      {
          // écoute de l'événement Event.ADDED_TO_STAGE
          addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );

      }

      private function ajoutAffichage( pEvt:Event ):void

      {

          // écoute de l'événement MouseEvent.CLICK
          stage.addEventListener ( MouseEvent.CLICK, clicSouris );

      }

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

```
            {

                  // affecte les coordonnées aux propriétés
                  sourisX = pEvt.stageX;
                  sourisY = pEvt.stageY;

            }

      }

}
```

Enfin, nous déclenchons le mouvement en écoutant l'événement `Event.ENTER_FRAME` hérité de la classe `Sprite` :

```
package

{

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

  // la classe Balle étend la classe Sprite
  public class Balle extends Sprite

  {

        // stocke les coordonnées de la souris
        private var sourisX:Number;
        private var sourisY:Number;

        public function Balle ()

        {
              // écoute de l'événement Event.ADDED_TO_STAGE
              addEventListener ( Event.ADDED_TO_STAGE, ajoutAffichage );

        }

        private function ajoutAffichage( pEvt:Event ):void

        {

              // écoute de l'événement MouseEvent.CLICK
              stage.addEventListener ( MouseEvent.CLICK, clicSouris );

        }

        private function clicSouris ( pEvt:MouseEvent ):void

        {

              // affecte les coordonnées aux propriétés
              sourisX = pEvt.stageX;
              sourisY = pEvt.stageY;

              addEventListener ( Event.ENTER_FRAME, mouvement );

        }
```

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

        {

              // évalue la destination x et y
              var destinationX:Number = ( sourisX - width/2);
              var destinationY:Number = ( sourisY - height/2);

              // déplace la balle avec un effet de ralentissement (inertie)
              x -= (x - destinationX)*.1;
              y -= (y - destinationY)*.1;

        }

  }

}
```

Si nous testons notre animation, la balle se déplace à l'endroit cliqué avec un effet de ralenti. La figure 10.1 illustre le comportement.

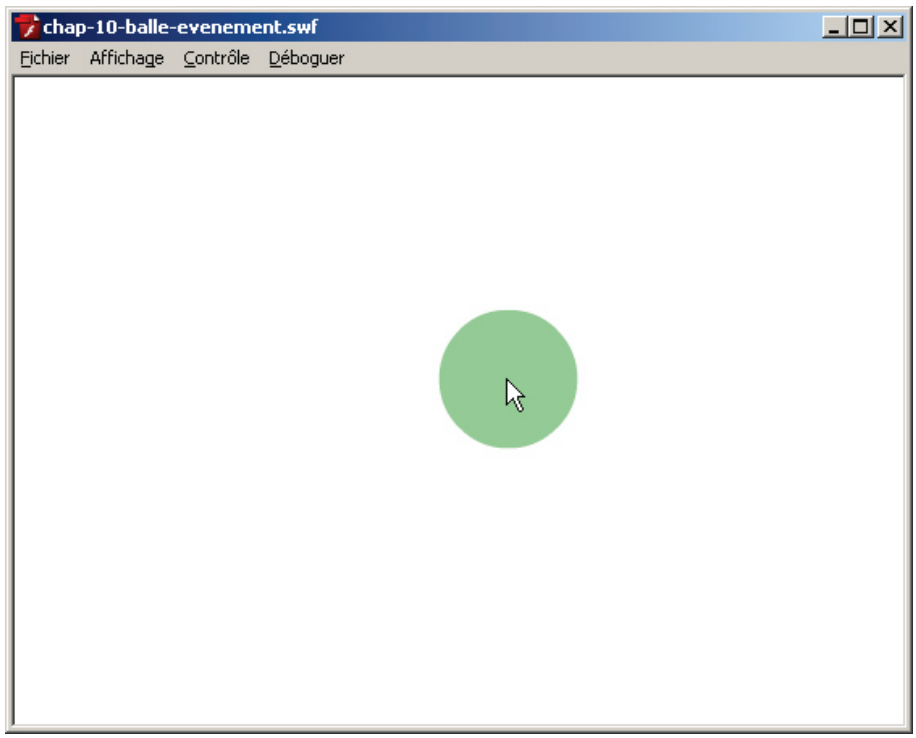

Figure 10.1 Déplacement position clic souris.

Lorsqu'un mouvement entre en jeu, nous souhaitons généralement savoir quand est ce qu'il se termine. La classe `Tween` diffuse par défaut tous les événements nécessaires à la synchronisation d'une animation. Mais comment faire dans notre cas pour diffuser notre propre événement ?

La classe `Sprite` hérite de la classe `EventDispatcher` et peut donc diffuser n'importe quel événement. Au sein de la méthode

`mouvement` nous testons si la différence entre la position en cours et la destination est inférieure à 1. Si c'est le cas, cela signifie que nous sommes arrivés à destination.

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

{

	// évalue la destination x et y
	var destinationX:Number = ( sourisX - width/2);
	var destinationY:Number = ( sourisY - height/2);

	// déplace la balle avec un effet de ralentissement (inertie)
	x -= (x - destinationX)*.1;
	y -= (y - destinationY)*.1;

	if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY
) < 1 )

	{

		removeEventListener ( Event.ENTER_FRAME, mouvement );

		trace ("arrivé à destination !");

	}

}
```

La méthode `Math.abs` nous permet de rendre la distance absolue, car une distance entre deux points est toujours positive.

Nous supprimons l'écoute de l'événement `Event.ENTER_FRAME` lorsque la balle arrive à destination afin d'optimiser les ressources, puis nous affichons un message indiquant que la balle est arrivée.

En testant notre animation, nous remarquons que le message indiquant l'arrivée est bien déclenché, mais pour l'instant aucun événement n'est diffusé.

Choisir un nom d'événement

Lorsqu'un objet diffuse un événement nous devons nous assurer que son nom soit *simple* et *intuitif* pour les personnes utilisant la classe. Dans notre exemple nous allons diffuser un événement `mouvementTermine`.

Nous modifions la méthode `mouvement` afin que celle-ci le diffuse :

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

{

	// évalue la destination x et y
	var destinationX:Number = ( sourisX - width/2);
	var destinationY:Number = ( sourisY - height/2);
```

```
        // déplace la balle avec un effet de ralentissement (inertie)
        x -= (x - destinationX)*.1;
        y -= (y - destinationY)*.1;

        if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY
) < 1 )

        {

             removeEventListener ( Event.ENTER_FRAME, mouvement );

             // diffusion de l'événement motionComplete
             dispatchEvent ( new Event ("mouvementTermine") );

        }

}
```

Notre symbole balle diffuse désormais un événement `mouvementTermine`. Il ne nous reste plus qu'à l'écouter :

```
// création du symbole
// affiche : [object Balle]
var maBalle:Balle = new Balle();

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

// écoute de l'événement personnalisé motionComplete
maBalle.addEventListener ( "mouvementTermine", arrivee );

// fonction écouteur
function arrivee ( pEvt:Event ):void

{

  trace("mouvement terminé !");

}
```

La fonction écouteur `arrivee` est déclenchée lorsque la balle arrive à destination. En cas de réutilisation de la classe `Balle` nous savons que celle-ci diffuse l'événement `mouvementTermine`. Libre à nous de décider quoi faire lorsque l'événement est diffusé, la réutilisation de la classe `Balle` est donc facilitée.

Notre code fonctionne très bien, mais il n'est pas totalement optimisé. Voyez-vous ce qui pose problème ?

Souvenez-vous, lors du chapitre 3 intitulé *Le modèle événementiel*, nous avons vu que nous ne qualifions **jamais** le nom des événements directement. Cela rend notre code rigide et non standard.

Nous préférons l'utilisation de constantes de classe. Nous définissons donc une propriété constante `MOUVEMENT_TERMINE` au sein de la classe `Balle` contenant le nom de l'événement diffusé :

```
// stocke le nom de l'événement diffusé
public static const MOUVEMENT_TERMINE:String = "mouvementTermine";
```

Puis nous ciblons la propriété afin d'écouter l'événement :

```
// écoute de l'événement personnalisé  Balle.MOUVEMENT_TERMINE
maBalle.addEventListener ( Balle.MOUVEMENT_TERMINE, arrivee );
```

Nous modifions la méthode mouvement afin de cibler la constante `Balle.MOUVEMENT_TERMINE` :

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

{

        // évalue la destination x et y
        var destinationX:Number = ( sourisX - width/2);
        var destinationY:Number = ( sourisY - height/2);

        // déplace la balle avec un effet de ralentissement (inertie)
        x -= (x - destinationX)*.1;
        y -= (y - destinationY)*.1;

        if ( Math.abs ( x - destinationX ) < 1 && Math.abs ( y - destinationY
) < 1 )

        {

             removeEventListener ( Event.ENTER_FRAME, mouvement );

             // diffusion de l'événement motionComplete
             dispatchEvent ( new Event ( Balle.MOUVEMENT_TERMINE ) );

        }

  }
```

Nous venons de traiter à travers cet exemple le cas le plus courant lors de la diffusion d'événements personnalisés. Voyons maintenant d'autres cas.

A retenir

- Un événement doit porter un nom simple et intuitif.
- Afin de stocker le nom d'un événement nous utilisons **toujours** une propriété constante de classe.

Etendre EventDispatcher

Rappelez-vous que toutes les classes résidant dans le paquetage `flash` héritent de la classe `EventDispatcher`. Dans vos développements, certaines classes n'hériteront pas de classes natives et n'auront pas la possibilité de diffuser des événements par défaut.

Prenons le cas d'une classe nommée `XMLLoader` devant charger des données XML. Nous souhaitons indiquer la fin du chargement des données en diffusant un événement approprié.

Le code suivant illustre le contenu de la classe :

```
package

{
  import flash.events.Event;

  public class XMLLoader

  {

        public function XMLLoader ()

        {

              // code non indiqué

        }

        public function charge ( pXML:String ):void

        {

              // code non indiqué

        }

        public function chargementTermine ( pEvt:Event ):void

        {

              dispatchEvent ( new Event ( XMLLoader.COMPLETE ) );

        }

  }


}
```

Si nous tentons de compiler cette classe, un message d'erreur nous avertira qu'aucune méthode `dispatchEvent` n'existe au sein de la classe. Afin d'obtenir les capacités de diffusion la classe `XMLLoader` hérite de la classe `EventDispatcher` :

```
package

{
  import flash.events.Event;
  import flash.events.EventDispatcher;

  public class XMLLoader extends EventDispatcher

  {

        public static const COMPLETE:String = "complete";
```

```
        public function XMLLoader ()
        {
            // code non indiqué
        }
        public function charge ( pXML:String ):void
        {
            // code non indiqué
        }
        public function chargementTermine ( pEvt:Event ):void
        {
            dispatchEvent ( new Event ( XMLLoader.COMPLETE ) );
        }
    }
}
```

En sous classant `EventDispatcher`, la classe `XMLLoader` peut désormais diffuser des événements. Afin d'écouter l'événement `XMLLoader.COMPLETE`, nous écrivons le code suivant :

```
// création d'une instance de XMLLoader
var chargeurXML:XMLLoader = new XMLLoader();

// chargement des données
chargeurXML.charge ("news.xml");

// écoute de l'événement XMLLoader.COMPLETE
chargeurXML.addEventListener ( XMLLoader.COMPLETE, chargementTermine );

// fonction écouteur
function chargementTermine ( pEvt:Event ):void
{
  trace( "données chargées");
}
```

Lorsque nous appelons la méthode `charge`, les données XML sont chargées. Une fois le chargement terminé nous diffusons l'événement `XMLLoader.COMPLETE`.

Une partie du code de la classe `XMLLoader` n'est pas montré car nous n'avons pas encore traité la notion de chargement externe. Nous

reviendrons sur cette classe au cours du chapitre 14 intitulé *Chargement et envoi de données* afin de la compléter.

Bien que cette technique soit efficace, elle ne révèle pas être la plus optimisée. Comme nous l'avons vu lors du chapitre 8 intitulé *Programmation orientée objet*, ActionScript n'intègre pas d'héritage multiple. Ainsi, en héritant de la classe `EventDispatcher` la classe `XMLLoader` ne peut hériter d'une autre classe. Nous cassons la chaîne d'héritage. Pire encore, si la classe `XMLLoader` devait étendre une autre classe, nous ne pourrions étendre en même temps `EventDispatcher`.

Dans un scénario comme celui-ci nous allons utiliser une notion essentielle de la programmation orientée objet abordée au cours du chapitre 8. Peut être avez vous déjà une idée ?

Stocker EventDispatcher

Afin de bien comprendre cette nouvelle notion, nous allons nous attarder sur la classe `Administrateur` développée lors du chapitre 8.

Voici le code de la classe `Administrateur` :

```
package

{

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

  {

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

       {

            super ( pPrenom, pNom, pAge, pVille );

       }

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

       {

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

            trace("Je suis modérateur");

       }

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

```
            {

                trace ("Kick " + pJoueur );

            }

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

            {

                trace("Joue un son");

            }

        }

    }
```

En découvrant la notion d'événements personnalisés nous décidons de diffuser un événement lorsqu'un joueur est supprimé de la partie. Il faudrait donc que la méthode `kickJoueur` puisse appeler la méthode `dispatchEvent`, ce qui est impossible pour le moment.

La classe `Administrateur` ne peut par défaut diffuser des événements, nous tentons donc d'étendre la classe `EventDispatcher`. Nous sommes alors confrontés à un problème, car la classe `Administrateur` étend déjà la classe `Joueur`.

Comment allons nous faire, sommes nous réellement bloqués ?

Souvenez vous, nous avons vu au cours du chapitre 8 une alternative à l'héritage. Cette technique décrivait une relation de type « possède un » au lieu d'une relation de type « est un ». En résumé, au lieu d'étendre une classe pour hériter de ses capacités nous allons créer une instance de celle-ci au sein de la classe et déléguer les fonctionnalités.

Ainsi au lieu de sous classer `EventDispatcher` nous allons stocker une instance de la classe `EventDispatcher` et déléguer la gestion des événements à celle-ci :

```
package

    {

        import flash.events.EventDispatcher;

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

        {

            // stocke l'instance d'EventDispatcher
            private var diffuseur:EventDispatcher;
```

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

        {

            super ( pPrenom, pNom, pAge, pVille );

            // création d'une instance d'EventDispatcher
            diffuseur = new EventDispatcher();

        }

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

        {

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

            trace("Je suis modérateur");

        }

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

        {

            // code gérant la déconnexion du joueur concerné
            trace ("Kick " + pJoueur );

        }

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

        {

            trace("Joue un son");

        }

    }

}
```

Bien entendu, la classe `Administrateur` ne possède pas pour le moment les méthodes `dispatchEvent`, `addEventListener`, et `removeEventListener`.

C'est à nous de les implémenter, pour cela nous implémentons l'interface `flash.events.IEventDispatcher` :

```
package

{

  import flash.events.EventDispatcher;
  import flash.events.IEventDispatcher;
  import flash.events.Event;
```

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

{

      // stocke l'instance d'EventDispatcher
      private var diffuseur:EventDispatcher;

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

      {

            super ( pPrenom, pNom, pAge, pVille );

            // création d'une instance d'EventDispatcher
            diffuseur = new EventDispatcher();

      }

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

      {

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

            trace("Je suis modérateur");

      }

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

      {

            // code gérant la déconnexion du joueur concerné

            trace ("Kick " + pJoueur );

      }

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

      {

            trace("Joue un son");

      }

      public function addEventListener( type:String, listener:Function,
useCapture:Boolean=false, priority:int=0, useWeakReference:Boolean=false
):void

      {

            diffuseur.addEventListener( type, listener, useCapture, priority,
useWeakReference );
```

```
        }

        public function dispatchEvent( event:Event ):Boolean

        {

             return diffuseur.dispatchEvent( event );

        }

        public function hasEventListener( type:String ):Boolean

        {

             return diffuseur.hasEventListener( type );

        }

        public function removeEventListener( type:String, listener:Function,
useCapture:Boolean=false ):void

        {

             diffuseur.removeEventListener( type, listener, useCapture );

        }

        public function willTrigger( type:String ):Boolean

        {

             return diffuseur.willTrigger( type );

        }

   }

}
```

Afin de rendre notre classe `Administrateur` diffuseur d'événements nous implémentons l'interface `IEventDispatcher`. Chacune des méthodes définissant le type `EventDispatcher` doivent donc être définies au sein de la classe `Administrateur`.

Nous définissons une constante de classe `KICK_JOUEUR`, stockant le nom de l'événement :

```
public static const KICK_JOUEUR:String = "deconnecteJoueur";
```

Puis nous modifions la méthode `kickJoueur` afin de diffuser un événement `Administrateur.KICK_JOUEUR` :

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

{

        // code gérant la déconnexion du joueur concerné

        // diffuse l'événement Administrateur.KICK_JOUEUR
```

```
            dispatchEvent ( new Event ( Administrateur.KICK_JOUEUR ) );

        }
```

Dans le code suivant, nous créons un modérateur et un joueur. Le joueur est supprimé de la partie, l'événement `Administrateur.KICK_JOUEUR` est bien diffusé :

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

// écoute l'événement Administrateur.KICK_JOUEUR
monModo.addEventListener (Administrateur.KICK_JOUEUR, deconnexionJoueur );

var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");

// un joueur est supprimé de la partie
monModo.kickJoueur ( premierJoueur );

// fonction écouteur
function deconnexionJoueur ( pEvt:Event ):void

{

  trace( "Un joueur a quitté la partie !" );

}
```

Grâce à la composition, nous avons pu rendre la classe `Administrateur` diffuseur d'événements. L'héritage n'est donc pas la seule alternative permettant de rendre une classe diffuseur d'événements.

A retenir

- Lorsque nous ne pouvons pas étendre la classe `EventDispatcher` nous stockons une instance de celle-ci et déléguons les fonctionnalités.
- L'interface `IeventDispatcher` permet une implémentation obligatoire des différentes méthodes nécessaires à la diffusion d'événements.

Passer des informations

La plupart des événements diffusés contiennent différentes informations relatives à l'état de l'objet diffusant l'événement. Nous avons vu lors du chapitre 6 intitulé *Intéractivité* que les classes `MouseEvent` ou `KeyboardEvent` possédaient des propriétés renseignant sur l'état de l'objet diffusant l'événement.

Dans l'exercice précédent, la classe `Administrateur` diffuse un événement `Administrateur.KICK_JOUEUR` mais celui-ci ne contient aucune information. Il serait intéressant que celui-ci nous renseigne sur le joueur supprimé. De la même manière la classe

`XMLLoader` pourrait diffuser un événement `XMLoader.COMPLETE` contenant le flux XML chargé afin qu'il soit facilement utilisable par les écouteurs.

Afin de passer des informations lors de la diffusion d'un événement nous devons étendre la classe `Event` afin de créer un objet événementiel spécifique à l'événement diffusé. En réalité, nous nous plions au modèle définit par ActionScript 3. Lorsque nous écoutons un événement lié à la souris nous nous dirigeons instinctivement vers la classe `flash.events.MouseEvent`. De la même manière pour écouter des événements liés au clavier nous utilisons la classe `flash.events.KeyboardEvent`.

De manière générale, il convient de créer une classe événementielle pour chaque classe devant diffuser des événements contenant des informations particulières.

Nous allons donc étendre la classe `Event` et créer une classe nommée `AdministrateurEvent` :

```
package

{

  import flash.events.Event;

  public class AdministrateurEvent extends Event

  {

        public static const KICK_JOUEUR:String = "deconnecteJoueur";

        public function AdministrateurEvent ( type:String,
bubbles:Boolean=false, cancelable:Boolean=false )

        {

              // initialisation du constructeur de la classe Event
              super( type, bubbles, cancelable );

        }

        // la méthode clone doit être surchargée
        public override function clone ():Event

        {

              return new AdministrateurEvent ( type, bubbles, cancelable )

        }

        // la méthode toString doit être surchargée
        public override function toString ():String

        {
```

```
            return '[AdministrateurEvent type="'+ type +'" bubbles=' +
bubbles + ' cancelable=' + cancelable + ']';
        }

    }

}
```

Puis nous utilisons une instance de cette classe afin de diffuser l'événement :

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

    {

        // code gérant la déconnexion du joueur concerné

        // diffuse l'évenement AdministrateurEvent.KICK_JOUEUR
        dispatchEvent ( new AdministrateurEvent (
AdministrateurEvent.KICK_JOUEUR ) );

    }
```

Il est important de noter que jusqu'à présent nous n'avions diffusé des événements qu'avec la classe `Event`. Lorsque nous définissons une classe événementielle spécifique nous stockons la constante de classe dans celle-ci.

Ainsi au lieu de cibler le nom de l'événement sur la classe diffusant l'événement :

```
// écoute l'événement Administrateur.KICK_JOUEUR
monModo.addEventListener ( Administrateur.KICK_JOUEUR, joueurQuitte );
```

Nous préférerons stocker le nom de l'événement au sein d'une constante de la classe événementielle :

```
// écoute l'événement AdministrateurEvent.KICK_JOUEUR
monModo.addEventListener ( AdministrateurEvent.KICK_JOUEUR, joueurQuitte );
```

A ce stade, aucune information ne transite par l'objet événementiel `AdministrateurEvent`. Afin de stocker des données supplémentaires nous définissons les propriétés voulues au sein de la classe puis nous affectons leur valeur selon les paramètres passés durant l'instanciation de l'objet événementiel :

```
package

{

    import flash.events.Event;

    public class AdministrateurEvent extends Event

    {

        public static const KICK_JOUEUR:String = "deconnecteJoueur";
        public var joueur:Joueur;
```

```
        public function AdministrateurEvent ( type:String,
bubbles:Boolean=false, cancelable:Boolean=false, pJoueur:Joueur=null )

        {

              // initialisation du constructeur de la classe Event
              super( type, bubbles, cancelable );

              // stocke le joueur supprimé
              joueur = pJoueur;

        }

        // la méthode clone doit être surchargée
        public override function clone ():Event

        {

              return new AdministrateurEvent ( type, bubbles, cancelable )

        }
        // la méthode toString doit être surchargée
        public override function toString ():String

        {

              return "[AdministrateurEvent type : " + type +",  bubbles : " +
bubbles + ", cancelable : " + cancelable + "]";

        }

  }

}
```

Puis nous modifions la méthode `kickJoueur` de la classe `Administrateur` afin de passer en paramètre le joueur supprimé :

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

  {

        // code gérant la déconnexion du joueur concerné

        // diffuse l'événement Administrateur.KICK_JOUEUR
        dispatchEvent ( new AdministrateurEvent ( AdministrateurEvent.
KICK_JOUEUR, false, false, pJoueur ) );

  }
```

Lorsque l'événement est diffusé, la fonction écouteur accède à la propriété `joueur` afin de savoir quel joueur a quitté la partie :

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

// écoute l'événement Administrateur.KICK_JOUEUR
monModo.addEventListener ( AdministrateurEvent.KICK_JOUEUR, joueurQuitte );

var premierJoueur:Joueur = new Joueur ("Bobby", "Womack", 66, "Detroit");
```

```
// un joueur est supprimé de la partie
monModo.kickJoueur ( premierJoueur );

// fonction écouteur
function joueurQuitte ( pEvt:AdministrateurEvent ):void

{

  // affiche : [AdministrateurEvent type="deconnecteJoueur" bubbles=false
cancelable=false]
  trace( pEvt );

  // affiche Bobby a quitté la partie !
  trace( pEvt.joueur.prenom + " a quitté la partie !");

}
```

En testant le code précédent, fonction écouteur `joueurQuitte` est notifié de l'événement `AdministrateurEvent.KICK_JOUEUR` et reçoit en paramètre un objet événementiel de type `AdministrateurEvent`.

La propriété `joueur` retourne le joueur supprimé de la partie, nous n'avons plus qu'à accéder aux propriétés voulues.

A retenir

- Afin de passer des paramètres lors de la diffusion d'un événement, nous devons obligatoirement étendre la classe `Event`.
- Les classes et sous-classes d'`Event` représentent les objets événementiels diffusés.

Menu et événement personnalisé

En fin de chapitre précédent nous avons développé un menu entièrement dynamique. Nous ne l'avions pas totalement finalisé car il nous manquait la notion d'événements personnalisés.

Souvenez-vous, nous avions besoin de diffuser un événement qui contiendrait le nom du SWF lié au bouton cliqué.

Au sein d'un répertoire `evenements` lui-même placé au sein du répertoire `org` nous créons une classe `ButtonEvent` :

```
package org.bytearray.evenements

{

  import flash.events.Event;

  public class ButtonEvent extends Event

  {
```

```
		public static const CLICK:String = "buttonClick";
		public var lien:String;

		public function ButtonEvent ( type:String, bubbles:Boolean=false,
cancelable:Boolean=false, pLien:String=null )

		{

			// initialisation du constructeur de la classe Event
			super( type, bubbles, cancelable );

			// stocke le lien lié au bouton cliqué
			lien = pLien;

		}

		// la méthode clone doit être surchargée
		public override function clone ():Event

		{

			return new ButtonEvent ( type, bubbles, cancelable, lien )

		}
		// la méthode toString doit être surchargée
		public override function toString ():String

		{

			return '[ButtonEvent type="'+ type +'" bubbles=' + bubbles + '
eventPhase='+ eventPhase + ' cancelable=' + cancelable + ']';

		}

	}

}
```

Dans le constructeur nous ajoutons la ligne suivante afin d'écouter l'événement `MouseEvent.CLICK` :

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

Nous définissons la méthode écouteur `clicSouris` qui diffuse l'événement `ButtonEvent.CLICK` en passant le SWF correspondant :

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

{

		// diffusion de l'événement ButtonEvent.CLICK
		dispatchEvent ( new ButtonEvent ( ButtonEvent.CLICK, true, false, swf
) );

}
```

Puis nous écoutons l'événement `ButtonEvent.CLICK` auprès de chaque bouton, nous utilisons la phase de capture afin d'optimiser le code :

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

// tableau associatif contenant les données
var donnees:Array = new Array();
donnees.push ( { legende : "Accueil", vitesse : 1, swf : "accueil.swf",
couleur : 0x999900 } );
donnees.push ( { legende : "Photos", vitesse : 1, swf : "photos.swf", couleur
: 0x881122 } );
donnees.push ( { legende : "Blog", vitesse : 1, swf : "blog.swf", couleur :
0x995471 } );
donnees.push ( { legende : "Liens", vitesse : 1, swf : "liens.swf", couleur :
0xCC21FF } );
donnees.push ( { legende : "Forum", vitesse : 1, swf : "forum.swf", couleur :
0x977821 } );

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

// conteneur du menu
var conteneurMenu:Sprite = new Sprite();

addChild ( conteneurMenu );

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

{

  // récupération des infos
  var legende:String = donnees[i].legende;
  var couleur:Number = donnees[i].couleur;
  var vitesse:Number = donnees[i].vitesse;
  var swf:String = donnees[i].swf;

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

  // positionnement
  monBouton.y = 50 * i;

  // ajout à la liste d'affichage
  conteneurMenu.addChild ( monBouton );

}

// écoute de l'événement ButtonEvent.CLICK pour la phase de capture
conteneurMenu.addEventListener ( ButtonEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:ButtonEvent ):void

{

  // affiche :
  /*accueil.swf
   photos.swf
```

```
   blog.swf
   liens.swf
   */
  trace( pEvt.lien );

}
```

Lorsque l'événement `ButtonEvent.CLICK` est diffusé, la fonction `clicBouton` est déclenchée. La propriété `lien` de l'objet événementiel de type `ButtonEvent` permet de charger le SWF correspondant. Celle-ci pourrait contenir dans une autre application une URL à atteindre lorsque l'utilisateur clique sur un bouton.

Il serait tout à fait envisageable de définir de nouvelles propriétés au sein de la classe `ButtonEvent` telles `legende`, `couleur` ou `vitesse` ou autres afin de passer les caractéristiques de chaque bouton.

Voici le code complet de la classe `Button` :

```
package org.bytearray.ui

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

  public class Button extends Sprite

  {

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

```
        public function Button ( pWidth:Number, pHeight:Number, pSWF:String,
pCouleur:Number, pVitesse:Number, pLegende:String )

        {

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

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

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

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

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

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

                // affecte la légende
                legende.text = pLegende;

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

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

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

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

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

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

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

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

                // stocke le nom du SWF
                swf = pSWF;

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

                // activation du mode bouton
                buttonMode = true;

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

```
            // affecte la vitesse passée en paramêtre
            setVitesse ( pVitesse );

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

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

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

      }

      private function clicSouris ( pEvt:MouseEvent ):void

      {

            // diffusion de l'événement ButtonEvent.CLICK
            dispatchEvent ( new ButtonEvent ( ButtonEvent.CLICK, true, false,
swf ) );

      }

      // déclenché lors du clic sur le bouton
      private function survolSouris ( pEvt:MouseEvent ):void

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

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

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

      }

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

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

      }

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

      {

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

            else

            {
```

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

            }

        }

    }

}
```

La diffusion d'événements personnalisés est un point essentiel dans tout développement orienté objet ActionScript. En diffusant nos propres événements nous rendons nos objets compatibles avec le modèle événementiel ActionScript 3 et facilement réutilisables.

En livrant une classe à un développeur tiers, celui-ci regardera en premier lieu les capacités offertes par celle-ci puis s'attardera sur les différents événements diffusés pour voir comment dialoguer facilement avec celle-ci.

A retenir

- Nous pouvons définir autant de propriétés que nous le souhaitons au sein de l'objet événementiel diffusé.

Nous allons nous intéresser maintenant à la notion de classe du document. Cette nouveauté apportée par Flash CS3 va nous permettre de rendre nos développements ActionScript 3 plus aboutis.

En avant pour le chapitre 11 intitulé *Classe du document*.