6

Propagation événementielle

Concept

Nous avons abordé la notion d'événements au cours du chapitre 3 intitulé *Modèle événementiel* et traité les nouveautés apportées par ActionScript 3. La propagation événementielle est un concept avancé d'ActionScript 3 qui nécessite une attention toute particulière. Nous allons au cours de ce chapitre apprendre à maîtriser ce nouveau comportement.

Le concept de propagation événementielle est hérité du *Document Object Model (DOM)* du W3C dont la dernière spécification est disponible à l'adresse suivante :

http://www.w3.org/TR/DOM-Level-3-Events/

En ActionScript 3, seuls les objets graphiques sont concernés par la propagation événementielle. Ainsi, lorsqu'un événement survient au sein de la liste d'affichage, le lecteur Flash propage ce dernier de l'objet `flash.display.Stage` jusqu'au parent direct de l'objet auteur de la propagation.

Cette descente de l'événement est appelée *phase de capture*.

Grâce à ce mécanisme, nous pouvons souscrire un écouteur auprès de l'un des parents de l'objet ayant initié la propagation, afin d'en être notifié durant la phase descendante. Le terme de capture est utilisé car nous pouvons *capturer* l'événement durant sa descente et stopper sa propagation si nécessaire.

La phase cible intervient lorsque l'événement a parcouru tous les objets parents et atteint l'objet ayant provoqué sa propagation, ce dernier est appelé *objet cible* ou *nœud cible*.

Une fois la phase cible terminée, le lecteur Flash propage l'événement dans le sens inverse de la phase de capture. Cette phase est appelée *phase de remontée*.

Durant celle-ci l'événement remonte jusqu'à l'objet `Stage` c'est-à-dire à la racine de la liste d'affichage.

Ces trois phases constituent *la propagation événementielle*, schématisée sur la figure 6-1 :

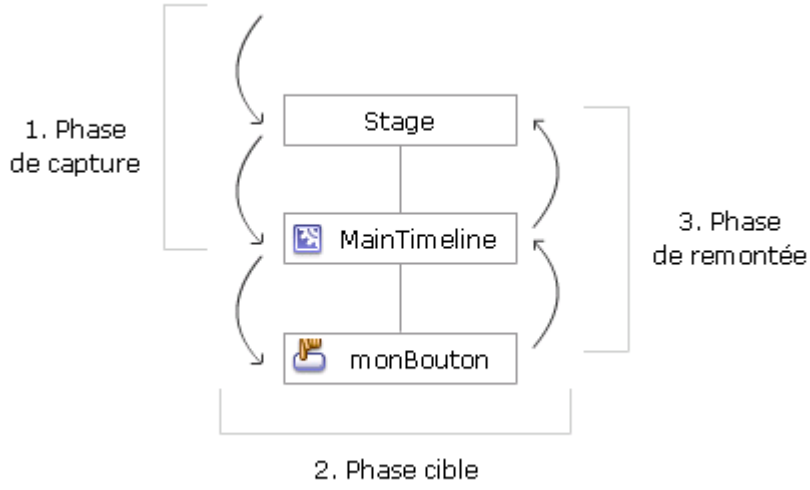

Figure 6-1. Les trois phases du flot événementiel.

Les événements diffusés en ActionScript 2 ne disposaient que d'une seule phase cible. Les deux nouvelles phases apportées par

ActionScript 3 vont nous permettre de mieux gérer l'interaction entre les objets graphiques au sein de la liste d'affichage.

> Attention, la propagation événementielle ne concerne que les objets graphiques.

Il est important de noter qu'ActionScript 3 n'est pas le seul langage à intégrer la notion de propagation événementielle, des langages comme JavaScript ou Java, intègrent ce mécanisme depuis plusieurs années déjà.

Nous allons nous attarder sur chacune des phases, et découvrir ensemble quels sont les avantages liés à cette propagation événementielle et comment en tirer profit dans nos applications.

A retenir

- La propagation événementielle ne concerne que les objets graphiques.
- La propagation événementielle n'est pas propre à ActionScript 3. D'autres langages comme JavaScript, Java ou C# l'intègre depuis plusieurs années déjà.
- Le flot événementiel se divise en trois phases distinctes : la phase de capture, la phase cible, et la phase de remontée.

La phase de capture

Comme nous l'avons abordé précédemment, la phase de capture est la première phase du flot événementiel.

En réalité, lorsque nous cliquons sur un bouton présent au sein de la liste d'affichage, tous les objets graphiques parents à ce dernier sont d'abord notifiés de l'événement et peuvent ainsi le diffuser.

La figure 6-2 illustre une situation classique, un bouton est posé sur le scénario principal :

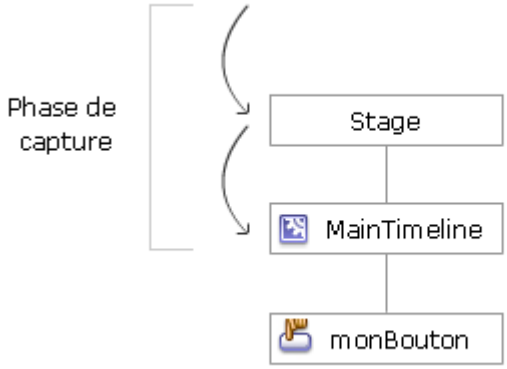

Figure 6-2. Phase de capture.

Lorsque l'utilisateur clique sur le bouton `monBouton`, la phase de capture démarre. Le lecteur Flash propage l'événement du haut vers le bas de la hiérarchie.

> Notons que la descente de l'événement s'arrête au parent direct du bouton, la phase suivante sera la phase cible.

L'objet `Stage`, puis l'objet `MainTimeline` (`root`) sont ainsi notifiés de l'événement. Ainsi, si nous écoutons l'événement en cours de propagation sur l'un de ces derniers, il nous est possible de l'intercepter.

En lisant la signature de la méthode `addEventListener` nous remarquons la présence d'un paramètre appelé `useCapture` :

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

Par défaut la méthode `addEventListener` ne souscrit pas l'écouteur passé à la phase de capture. Pour en profiter nous devons passer la valeur booléenne `true` au troisième paramètre nommé `useCapture`.

L'idée est d'écouter l'événement sur l'un des objets parents à l'objet cible :

```
objetParent.addEventListener ( MouseEvent.CLICK, clicBouton, true );
```

Passons à un peu de pratique, dans un nouveau document Flash CS3 nous créons un symbole de type bouton et nous posons une occurrence de ce dernier, appelée `monBouton`, sur la scène principale.

Dans le code suivant nous souscrivons un écouteur auprès du scénario principal en activant la capture :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
{

	// affiche : [MouseEvent type="click" bubbles=true cancelable=false
eventPhase=1 localX=13 localY=13 stageX=75.95 stageY=92 relatedObject=null
ctrlKey=false altKey=false shiftKey=false delta=0]
	trace( pEvt );
}
```

Lors du clic sur notre bouton `monBouton`, l'événement `MouseEvent.CLICK` entame sa phase de descente puis atteint le scénario principal qui le diffuse aussitôt. La fonction écouteur `clicBouton` est déclenchée.

Contrairement à la méthode `hasEventListener`, la méthode `willTrigger` permet de déterminer si un événement spécifique est écouté auprès de l'un des parents lors de la phase de capture ou de remontée :

```
 // souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
{

	trace( pEvt );
}

// aucun écouteur n'est enregistré auprès du bouton
// affiche : false
trace( monBouton.hasEventListener (MouseEvent.CLICK) );

// un écouteur est enregistré auprès d'un des parents du bouton
// affiche : true
trace( monBouton.willTrigger( MouseEvent.CLICK ) );

// un écouteur est enregistré auprès du scénario principal
// affiche : true
trace( willTrigger( MouseEvent.CLICK ) );
```

Nous préférerons donc l'utilisation de la méthode `willTrigger` dans un contexte de propagation événementielle.

Au cours du chapitre 3 intitulé *Modèle événementiel* nous avons découvert la propriété `target` de l'objet événementiel correspondant à l'objet auteur du flot événementiel que nous appelons généralement *objet cible*.

Attention, durant la phase de capture ce dernier n'est pas celui qui a diffusé l'événement, mais celui qui est à la *source de la propagation* événementielle. Dans notre cas, l'objet `MainTimeline` a diffusé

l'événement `MouseEvent.CLICK`, suite à un déclenchement de la propagation de l'événement par notre bouton.

L'intérêt majeur de la phase de capture réside donc dans la possibilité d'intercepter depuis un parent n'importe quel événement provenant d'un enfant.

La notion de nœuds

Lorsqu'un événement se propage, les objets graphiques notifiés sont appelés *objets nœuds*. Pour récupérer le nœud sur lequel se trouve l'événement au cours de sa propagation, nous utilisons la propriété `currentTarget` de l'objet événementiel.

> La propriété `currentTarget` renvoie *toujours* l'objet sur lequel nous avons appelé la méthode `addEventListener`.

Grace aux propriétés `target` et `currentTarget` nous pouvons donc savoir vers quel objet graphique se dirige l'événement ainsi que le nœud traité actuellement.

La figure 6-3 illustre l'idée :

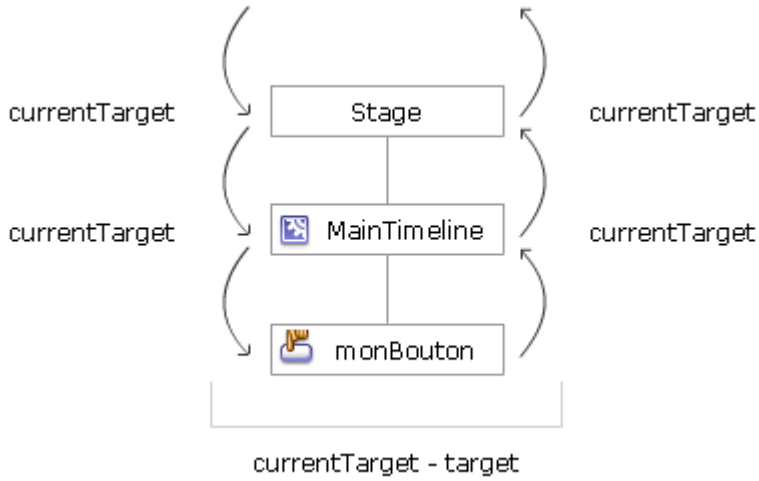

Figure 6-3. Propagation de l'événement au sein des objets nœuds.

En testant le code suivant, nous voyons que le scénario principal est notifié de l'événement `MouseEvent.CLICK` se dirigeant vers l'objet cible :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
```

```
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
{
	/*
	// affiche :
	Objet cible : [object SimpleButton]
	Noeud en cours : [object MainTimeline]
	*/
	trace( "Objet cible : " + pEvt.target )
	trace( "Noeud en cours : " + pEvt.currentTarget );
}
```

Bien que nous connaissions les objets graphiques associés à la propagation de l'événement, nous ne pouvons pas déterminer pour le moment la phase en cours.

L'événement actuellement diffusé correspond t'il à la phase de capture, cible, ou bien de remontée ?

Déterminer la phase en cours

Afin de savoir à quelle phase correspond un événement nous utilisons la propriété `eventPhase` de l'objet événementiel.

Durant la phase de capture, la propriété `eventPhase` vaut 1, 2 pour la phase cible et 3 pour la phase de remontée.

Dans le même document que précédemment nous écoutons l'événement `MouseEvent.CLICK` auprès du scénario principal durant la phase de capture :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )

{

	// affiche : Phase en cours : 1
	trace ( "Phase en cours : " + pEvt.eventPhase );

}
```

Lorsque nous cliquons sur notre bouton, la propriété `eventPhase` de l'objet événementiel vaut 1.

Pour afficher une information relative à la phase en cours nous pourrions être tentés d'écrire le code suivant :

```
// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicBouton, true );

function clicBouton ( pEvt:MouseEvent )
```

```
{

        // affiche : Phase en cours : 1
        if ( pEvt.eventPhase == 1 ) trace("PHASE DE CAPTURE");

}
```

Attention, comme pour la souscription d'événements, nous utilisons toujours des constantes de classes pour déterminer la phase en cours.

Pour cela, la classe `flash.events.EventPhase` contient trois propriétés associées à chacune des phases. En testant la valeur de chaque propriété nous récupérons les valeurs correspondantes :

```
// affiche : 1
trace( EventPhase.CAPTURING_PHASE );

// affiche : 2
trace( EventPhase.AT_TARGET);

// affiche : 3
trace( EventPhase.BUBBLING_PHASE);
```

La figure 6-4 illustre les constantes de la classe `EventPhase` liées à chaque phase :

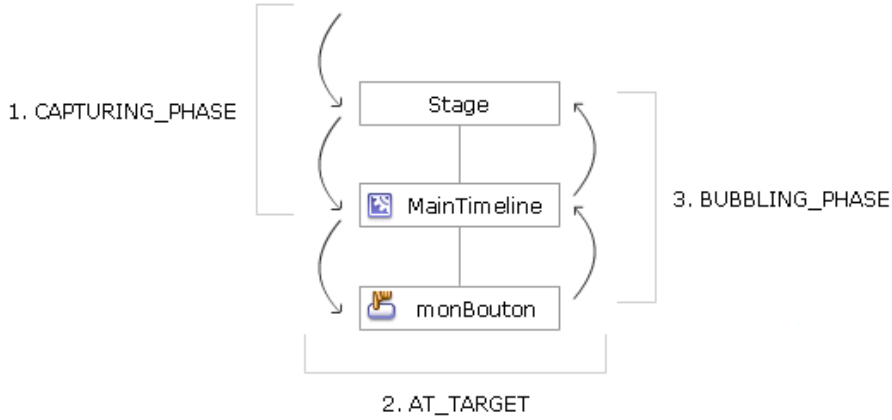

Figure 6-4. Les trois phases de la propagation événementielle.

En prenant cela en considération, nous pouvons réécrire la fonction écouteur `clicBouton` de la manière suivante :

```
function clicBouton ( pEvt:MouseEvent )

{

        switch ( pEvt.eventPhase )

        {

             case EventPhase.CAPTURING_PHASE :
```

```
                trace("phase de capture");
                break;

            case EventPhase.AT_TARGET :
                trace("phase cible");
                break;

            case EventPhase.BUBBLING_PHASE :
                trace("phase de remontée");
                break;

        }

    }
```

Tout cela reste relativement abstrait et nous pouvons nous demander l'intérêt d'un tel mécanisme dans le développement d'applications ActionScript 3.

Grâce à ce processus nous allons rendre notre code souple et optimisé. Nous allons en tirer profit au sein d'une application dans la partie suivante.

A retenir

- La phase de capture démarre de l'objet `Stage` jusqu'au parent de l'objet cible.
- La phase de capture ne concerne que les objets parents.
- La propriété `target` de l'objet événementiel renvoie toujours une référence vers l'objet cible.
- La propriété `currentTarget` de l'objet événementiel renvoie toujours une référence vers l'objet sur lequel nous avons appelé la méthode `addEventListener`.
- Durant la propagation d'un événement, la propriété `target` ne change pas et fait toujours référence au même objet graphique (objet cible).
- La propriété `eventPhase` de l'objet événementiel renvoie une valeur allant de 1 à 3 associées à chaque phase : CAPTURING_PHASE, AT_TARGET et BUBBLING_PHASE.
- Pour déterminer la phase liée à un événement nous comparons la propriété `eventPhase` aux trois constantes de la classe `flash.events.EventPhase`.

Optimiser le code avec la phase de capture

Afin de mettre en évidence l'intérêt de la phase de capture, nous allons prendre un cas simple d'application ActionScript où nous souhaitons écouter l'événement `MouseEvent.CLICK` auprès de différents

boutons. Lorsque nous cliquons sur chacun d'entre eux, nous les supprimons de la liste d'affichage.

Dans un nouveau document Flash CS3 nous créons un symbole bouton de forme rectangulaire lié à une classe `Fenetre` par le panneau *Propriétés de liaison*.

Puis nous ajoutons plusieurs instances de celle-ci à notre scénario principal :

```
// nombre de fenêtres
var lng:int = 12;

var maFenetre:Fenetre;

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

	maFenetre = new Fenetre();

	maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
	maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

	addChild ( maFenetre );

}
```

Nous obtenons le résultat illustré par la figure 6-5 :

Figure 6-5. Instances de la classe `Fenetre`.

Notre liste d'affichage pourrait être représentée de la manière suivante :

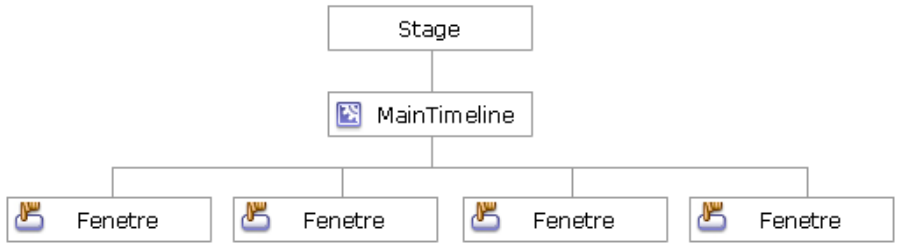

Figure 6-6. Liste d'affichage avec instances de la classe `Fenetre`.

Chaque clic sur une instance de symbole `Fenetre` provoque une propagation de l'événement `MouseEvent.CLICK` de l'objet `Stage` jusqu'à notre scénario principal `MainTimeline`.

Ce dernier est donc notifié de chaque clic sur nos boutons durant la phase de capture.

Comme nous l'avons vu précédemment, la phase de capture présente l'intérêt de pouvoir intercepter un événement durant sa phase descendante auprès d'un objet graphique parent.

Cela nous permet donc de centraliser l'écoute de l'événement `MouseEvent.CLICK` en appelant une seule fois la méthode `addEventListener`.

De la même manière, pour arrêter l'écoute d'un événement, nous appelons la méthode `removeEventListener` sur l'objet graphique parent seulement.

Ainsi nous n'avons pas besoin de souscrire un écouteur auprès de chaque instance de la classe `Fenetre` mais seulement auprès du scénario principal :

```
// nombre de fenêtres
var lng:int = 12;

var maFenetre:Fenetre;

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

{

        maFenetre = new Fenetre();

        maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
        maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

        addChild ( maFenetre );
```

```
}

// souscription à l'événement MouseEvent.CLICK auprès
// du scénario principal pour la phase de capture
addEventListener ( MouseEvent.CLICK, clicFenetre, true );

function clicFenetre ( pEvt:MouseEvent ):void

{

  // affiche : [MouseEvent type="click" bubbles=true cancelable=false
eventPhase=1 localX=85 localY=15 stageX=92 stageY=118 relatedObject=null
ctrlKey=false altKey=false shiftKey=false delta=0]
  trace( pEvt );

}
```

Si nous n'avions pas utilisé la phase de capture, nous aurions du souscrire un écouteur auprès de chaque symbole `Fenetre`.

En ajoutant un écouteur auprès de l'objet parent nous pouvons capturer l'événement provenant des objets enfants, rendant ainsi notre code centralisé. Si les boutons viennent à être supprimés nous n'avons pas besoin d'appeler la méthode `removeEventListener` sur chacun d'entre eux afin de libérer les ressources, car seul le parent est écouté.

Si par la suite les boutons sont recréés, aucun code supplémentaire n'est nécessaire.

Attention, en capturant l'événement auprès du scénario principal nous écoutons tous les événements `MouseEvent.CLICK` des objets interactifs enfants. Si nous souhaitons seulement écouter les événements provenant des instances de la classe `Fenetre`, il nous faut alors les isoler dans un objet conteneur, et procéder à la capture des événements auprès de ce dernier.

Nous préfèrerons donc l'approche suivante :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

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

var maFenetre:Fenetre;

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

{

       maFenetre = new Fenetre();

       maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
```

```
            maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

            conteneurFenetres.addChild ( maFenetre );

        }

        // souscription à l'événement MouseEvent.CLICK auprès
        // du conteneur pour la phase de capture
        conteneurFenetres.addEventListener ( MouseEvent.CLICK, clicFenetre, true );

        function clicFenetre ( pEvt:MouseEvent ):void

        {

          // affiche : [MouseEvent type="click" bubbles=true cancelable=false
        eventPhase=1 localX=119 localY=46 stageX=126 stageY=53 relatedObject=null
        ctrlKey=false altKey=false shiftKey=false delta=0]
          trace( pEvt );

        }
```

Afin de supprimer de l'affichage chaque fenêtre cliquée, nous ajoutons l'instruction `removeChild` au sein de la fonction écouteur `clicFenetre` :

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

        {

          // l'instance de Fenetre cliquée est supprimée de l'affichage
          pEvt.currentTarget.removeChild ( DisplayObject ( pEvt.target ) );

        }
```

La propriété `target` fait ainsi référence à l'instance de `Fenetre` cliquée, tandis que la propriété `currentTarget` référence le conteneur.

A chaque clic, l'instance cliquée est supprimée de la liste d'affichage :

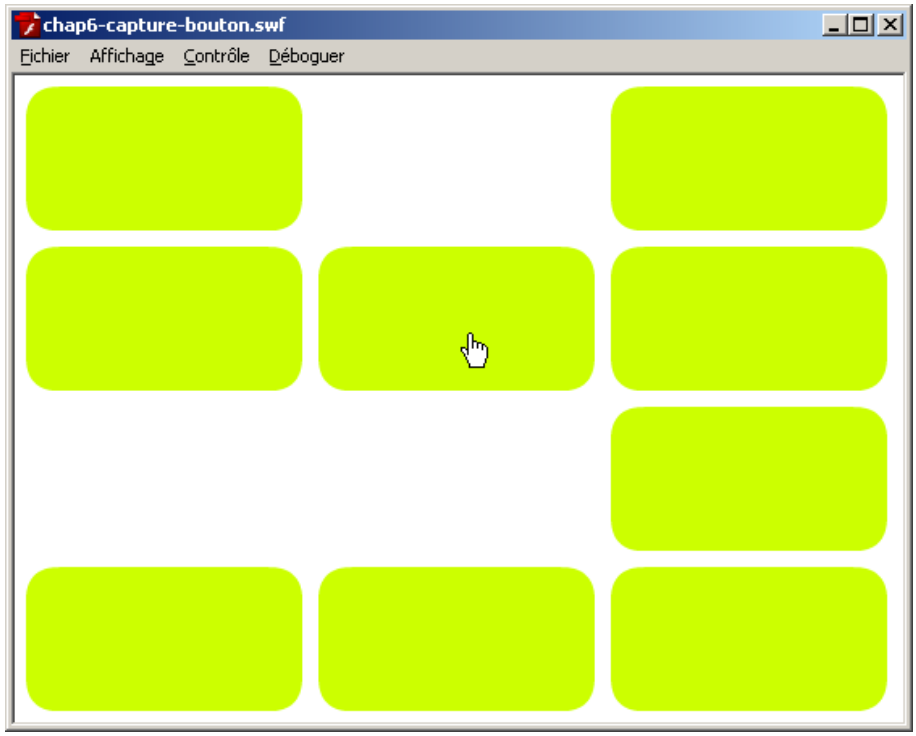

Figure 6-7. Suppression d'instances de la classe `Fenetre`.

Nous reviendrons tout au long de l'ouvrage sur l'utilisation de la phase de capture afin de maîtriser totalement ce concept.

Nous allons nous attarder à présent sur la phase cible.

A retenir

- Grâce à la phase de capture, nous souscrivons l'écouteur auprès de l'objet graphique parent afin de capturer les événements des objets enfants.
- Notre code est plus optimisé et plus centralisé.

La phase cible

La phase cible correspond à la diffusion de l'événement depuis *l'objet cible*. Certains événements associés à des objets graphiques ne participent qu'à celle-ci.

C'est le cas des quatre événements suivants qui ne participent ni à la phase de capture ni à la phase de remontée :

- `Event.ENTER_FRAME`
- `Event.ACTIVATE`

- `Event.DEACTIVATE`
- `Event.RENDER`

Nous allons reprendre l'exemple précédent, en préférant cette fois la phase cible à la phase de capture :

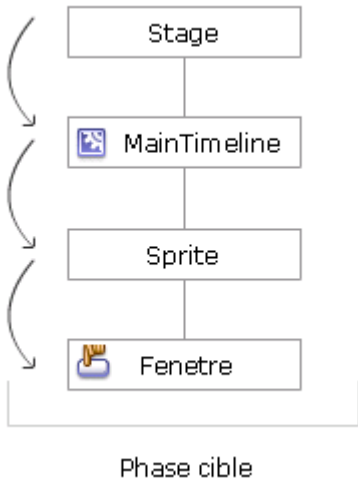

Figure 6-8. Phase cible.

Afin d'écouter l'événement `MouseEvent.CLICK`, nous souscrivons un écouteur auprès de chaque instance de la classe `Fenetre` :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

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

var maFenetre:Fenetre;

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

{

	maFenetre = new Fenetre();

	// souscription auprès de chaque instance pour la phase cible
	maFenetre.addEventListener ( MouseEvent.CLICK, clicFenetre );

	maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
	maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

	conteneurFenetres.addChild ( maFenetre );

}

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

```
{

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

}
```

Au clic, l'événement commence sa propagation puis atteint l'objet cible, notre fonction écouteur `clicFenetre` est déclenchée.

> Ainsi, lorsque nous appelons la méthode `addEventListener` sans spécifier le paramètre `useCapture`, nous souscrivons l'écouteur à la phase cible ainsi qu'à la phase de remontée que nous traiterons juste après.

Si nous affichons le contenu de la propriété `target` de l'objet événementiel, celui-ci nous renvoie une référence vers l'objet cible, ici notre objet `Fenetre`.

Pour supprimer chaque instance, nous rajoutons l'instruction `removeChild` au sein de notre fonction écouteur en passant l'objet référencé par la propriété `target` de l'objet événementiel :

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

{

  // l'instance de Fenetre cliquée est supprimée de l'affichage
  conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

  // désinscription de la fonction clicFenetre à l'événement
  // MouseEvent.CLICK
  pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );

}
```

Il est important de noter que durant la phase cible, l'objet cible est aussi le diffuseur de l'événement.

En testant le code suivant, nous voyons que les propriétés `target` et `currentTarget` de l'objet événementiel référencent le même objet graphique :

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

{

  // affiche : true
  trace( pEvt.target == pEvt.currentTarget );

  // l'instance de Fenetre cliquée est supprimée de l'affichage
  conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

  // désinscription de la fonction clicFenetre à l'événement
  // MouseEvent.CLICK
```

```
        pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );

    }
```

Pour tester si l'événement diffusé est issu de la phase cible nous pouvons comparer la propriété `eventPhase` de l'objet événementiel à la constante `EventPhase.AT_TARGET` :

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

    {

        // affiche : true
        trace( pEvt.target == pEvt.currentTarget );

        // affiche : PHASE CIBLE
        if ( pEvt.eventPhase == EventPhase.AT_TARGET )

        {

            trace("phase cible");

        }

        // l'instance de Fenetre cliquée est supprimée de l'affichage
        conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

        // désinscription de la fonction clicFenetre à l'événement
        // MouseEvent.CLICK
        pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );

    }
```

En utilisant la phase cible nous avons perdu en souplesse en souscrivant la fonction écouteur `clicFenetre` auprès de chaque instance de la classe `Fenetre`, et géré la désinscription des écouteurs lors de la suppression de la liste d'affichage.

Bien entendu, la phase de capture ne doit et ne peut pas être utilisée systématiquement. En utilisant la phase cible nous pouvons enregistrer une fonction écouteur à un bouton unique, ce qui peut s'avérer primordial lorsque la logique associée à chaque bouton est radicalement différente. Bien que très pratique dans certains cas, la phase de capture intercepte les clics de chaque bouton en exécutant une seule fonction écouteur ce qui peut s'avérer limitatif dans certaines situations.

Chacune des phases doit donc être considérée et utilisée lorsque la situation vous paraît la plus appropriée.

A retenir

- La phase cible correspond à la diffusion de l'événement par l'objet cible.
- Les objets graphiques diffusent des événements pouvant se propager.
- Les objets non graphiques diffusent des événements ne participant qu'à la phase cible.
- Il convient de bien étudier l'imbrication des objets graphiques au sein de l'application, et d'utiliser la phase la plus adaptée à nos besoins.

ActionScript 3 offre en plus la possibilité d'intervenir sur la propagation d'un événement. Cette fonctionnalité peut être utile lorsque nous souhaitons empêcher un événement d'atteindre l'objet cible en empêchant la poursuite de sa propagation.

Nous allons découvrir à l'aide d'un exemple concret comment utiliser cette nouvelle notion.

Intervenir sur la propagation

Il est possible de stopper la propagation d'un événement depuis n'importe quel nœud. Quel pourrait être l'intérêt de stopper la propagation d'un événement durant sa phase de capture ou sa phase de remontée ?

Dans certains cas nous avons besoin de verrouiller l'ensemble d'une application, comme par exemple la sélection d'éléments interactifs dans une application ou dans un jeu.

Deux méthodes définies par la classe `flash.events.Event` permettent d'intervenir sur la propagation :

- `objtEvenementiel.stopPropagation()`
- `objtEvenementiel.stopImmediatePropagation()`

Ces deux méthodes sont quasiment identiques mais diffèrent quelque peu, nous allons nous y attarder à présent.

Imaginons que nous souhaitions verrouiller tous les boutons de notre exemple précédent.

Dans l'exemple suivant nous allons intervenir sur la propagation d'un événement `MouseEvent.CLICK` durant la phase de capture. En stoppant sa propagation au niveau du conteneur, nous allons l'empêcher d'atteindre l'objet cible.

Ainsi la phase cible ne sera jamais atteinte.

La figure 6-9 illustre l'interruption de propagation d'un événement :

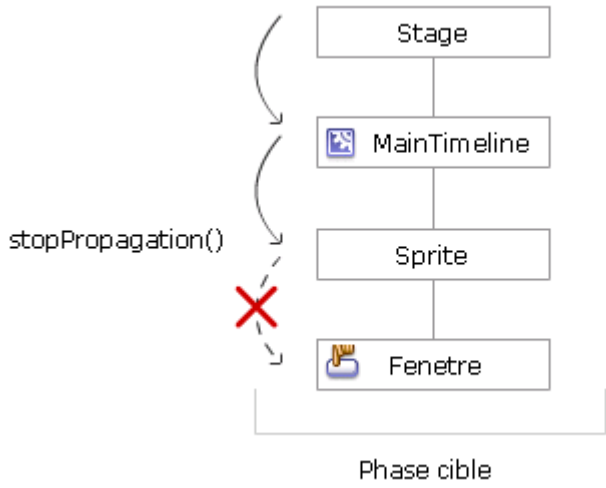

Figure 6-9. Intervention sur la propagation d'un événement.

Afin d'intercepter l'événement `MouseEvent.CLICK` avant qu'il ne parvienne jusqu'à l'objet cible, nous écoutons l'événement `MouseEvent.CLICK` lors de la phase de capture auprès du conteneur `conteneurFenetres` :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

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

var maFenetre:Fenetre;

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

{

	maFenetre = new Fenetre();

	// souscription auprès de chaque instance pour la phase cible
	maFenetre.addEventListener ( MouseEvent.CLICK, clicFenetre );

	maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
	maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

	conteneurFenetres.addChild ( maFenetre );

}

// souscription à l'événement MouseEvent.CLICK auprès
// du conteneur pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );
```

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

{

  // l'instance de Fenetre cliquée est supprimée de l'affichage
  conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

  // désinscription de la fonction clicFenetre à l'événement
  // MouseEvent.CLICK
  pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );

}

function captureClic ( pEvt:MouseEvent ):void

{

  // affiche : Capture de l'événement : click
  trace("Capture de l'événement : " + pEvt.type );

}
```

Nous remarquons que la fonction `captureClic` est bien déclenchée à chaque clic bouton.

Pour stopper la propagation, nous appelons la méthode `stopPropagation` sur l'objet événementiel diffusé :

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

{

  // affiche : Objet cible : [object Fenetre]
  trace( "Objet cible : " + pEvt.target );

  // affiche : Objet notifié : [object Sprite]
  trace( "Objet notifié : " + pEvt.currentTarget );

  // affiche : Capture de l'événement : click
  trace("Capture de l'événement : " + pEvt.type );

  pEvt.stopPropagation();

}
```

Lorsque la méthode `stopPropagation` est exécutée, l'événement interrompt sa propagation sur le nœud en cours, la fonction `clicFenetre` n'est plus déclenchée.

Grâce au code suivant, les instances de la classe `Fenetre` sont supprimées uniquement lorsque la touche ALT est enfoncée :

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

{

  // affiche : Objet cible : [object Fenetre]
  trace( "Objet cible : " + pEvt.target );
```

```
        // affiche : Objet notifié : [object Sprite]
        trace( "Objet notifié : " + pEvt.currentTarget );

        // affiche : Capture de l'événement : click
        trace("Capture de l'événement : " + pEvt.type );

        if ( !pEvt.altKey ) pEvt.stopPropagation();

    }
```

Nous reviendrons sur les propriétés de la classe `MouseEvent` au cours du prochain chapitre intitulé *Interactivité*.

Attention, la méthode `stopPropagation` interrompt la propagation de l'événement auprès des nœuds suivants, mais autorise l'exécution des écouteurs souscrits au nœud en cours. Si nous ajoutons un autre écouteur à l'objet `conteneurFenetres`, bien que la propagation soit stoppée, la fonction écouteur est déclenchée.

Afin de mettre en évidence la méthode `stopImmediatePropagation` nous ajoutons une fonction `captureClicBis` comme autre écouteur du conteneur :

```
    // nombre de fenêtres
    var lng:int = 12;

    // création d'un conteneur
    var conteneurFenetres:Sprite = new Sprite();

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

    var maFenetre:Fenetre;

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

    {

            maFenetre = new Fenetre();

            // souscription auprès de chaque instance pour la phase cible
            maFenetre.addEventListener ( MouseEvent.CLICK, clicFenetre );

            maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
            maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

            conteneurFenetres.addChild ( maFenetre );

    }

    // souscription à l'événement MouseEvent.CLICK auprès
    // du conteneur pour la phase de capture
    conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );

    // souscription à l'événement MouseEvent.CLICK auprès
    // du conteneur pour la phase de capture
    conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClicBis, true
    );
```

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

{

  // l'instance de Fenetre cliquée est supprimée de l'affichage
  conteneurFenetres.removeChild ( DisplayObject ( pEvt.target ) );

  // désinscription de la fonction clicFenetre à l'événement
  // MouseEvent.CLICK
  pEvt.target.removeEventListener ( MouseEvent.CLICK, clicFenetre );

}

function captureClic ( pEvt:MouseEvent ):void

{

  // affiche : Objet cible : [object Fenetre]
  trace( "Objet cible : " + pEvt.target );
  // affiche : Objet notifié : [object Sprite]
  trace( "Objet notifié : " + pEvt.currentTarget );
  // affiche : Capture de l'événement : click
  trace("Capture de l'événement : " + pEvt.type );

  if ( !pEvt.altKey ) pEvt.stopPropagation();

}

function captureClicBis ( pEvt:MouseEvent ):void

{

  trace("fonction écouteur captureClicBis déclenchée");

}
```

En testant le code précédent, nous voyons que notre fonction `captureClicBis` est bien déclenchée bien que la propagation de l'événement soit interrompue. A l'inverse, si nous appelons la méthode `stopImmediatePropagation`, la fonction `captureClicBis` n'est plus déclenchée :

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

{

  // affiche : Objet cible : [object Fenetre]
  trace( "Objet cible : " + pEvt.target );

  // affiche : Objet notifié : [object Sprite]
  trace( "Objet notifié : " + pEvt.currentTarget );

  // affiche : Capture de l'événement : click
  trace("Capture de l'événement : " + pEvt.type );

  if ( !pEvt.altKey ) pEvt.stopImmediatePropagation();

}
```

Grâce aux méthodes `stopPropagation` et `stopImmediatePropagation` nous pouvons ainsi maîtriser le flot événementiel avec précision.

A retenir

- Il est possible d'intervenir sur la propagation d'un événement à l'aide des méthodes `stopPropagation` et `stopImmediatePropagation`.
- La méthode `stopPropagation` interrompt la propagation d'un événement mais n'empêche pas sa diffusion auprès des écouteurs du nœud en cours.
- La méthode `stopImmediatePropagation` interrompt la propagation d'un événement et empêche sa diffusion même auprès des écouteurs du même nœud.

La phase de remontée

La phase de remontée constitue la dernière phase de la propagation. Durant cette phase, l'événement parcourt le chemin inverse de la phase de capture, notifiant chaque nœud parent de l'objet cible.

Le parcourt de l'événement durant cette phase s'apparente à celui d'une bulle remontant à la surface de l'eau. C'est pour cette raison que cette phase est appelée en anglais « *bubbling phase* ». L'événement remontant de l'objet cible jusqu'à l'objet `Stage`.

La figure 6-10 illustre la phase de remontée :

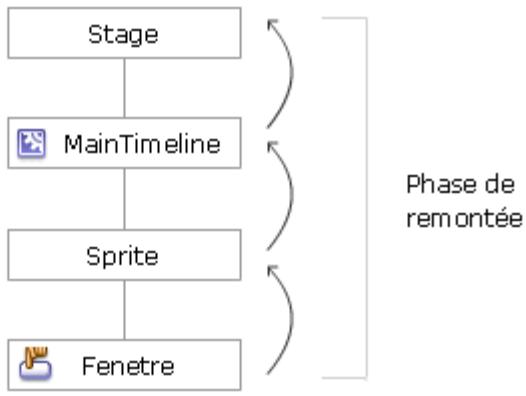

Figure 6-10. Phase de remontée.

Pour souscrire un écouteur auprès de la phase de remontée nous appelons la méthode `addEventListener` sur un des objets

graphiques parent en passant la valeur booléenne `false` au paramètre `useCapture` :

```
monObjetParent.addEventListener ( MouseEvent.CLICK, clicRemontee, false );
```

En lisant la signature de la méthode `addEventListener` nous voyons que le paramètre `useCapture` vaut `false` par défaut :

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

Ainsi, la souscription auprès de la phase de remontée peut s'écrire sans préciser le paramètre `useCapture` :

```
objetParent.addEventListener ( MouseEvent.CLICK, clicRemontee );
```

Cela signifie que lorsque nous écoutons un événement pour la phase cible, la phase de remontée est automatiquement écoutée.

La fonction `clicRemontee` recevra donc aussi les événements `MouseEvent.CLICK` provenant des enfants durant leur phase de remontée.

Dans l'exemple suivant, nous écoutons l'événement `MouseEvent.CLICK` lors de la phase de capture et de remontée auprès du conteneur :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

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

var maFenetre:Fenetre;

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

{

	maFenetre = new Fenetre();

	maFenetre.x = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
	maFenetre.y = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10 );

	conteneurFenetres.addChild ( maFenetre );

}

// souscription auprès du conteneur  pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );

// souscription auprès du conteneur pour la phase de remontée
conteneurFenetres.addEventListener ( MouseEvent.CLICK, clicRemontee );
```

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

{

  /* affiche :
  phase de capture de l'événement : click
  noeud en cours de notification : [object Sprite]
  */
  if ( pEvt.eventPhase == EventPhase.CAPTURING_PHASE )

  {

        trace("phase de capture de l'événement : " + pEvt.type );
        trace("noeud en cours de notification : " + pEvt.currentTarget );

  }

}

function clicRemontee ( pEvt:MouseEvent ):void

{

  /* affiche :
  phase de remontée de l'événement : click
  noeud en cours de notification : [object Sprite]
  */
  if ( pEvt.eventPhase == EventPhase.BUBBLING_PHASE )

  {

        trace("phase de remontée de l'événement : " + pEvt.type );
        trace("noeud en cours de notification : " + pEvt.currentTarget );

  }

}
```

Nous allons aller plus loin en ajoutant une réorganisation automatique des fenêtres lorsqu'une d'entre elles est supprimée.

Pour cela nous modifions le code en intégrant un effet d'inertie afin de disposer chaque instance de la classe `Fenetre` avec un effet de ralenti :

```
// modification de la cadence de l'animation
stage.frameRate = 30;

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

{

       maFenetre = new Fenetre();

       maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
       maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10
);

       maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

       conteneurFenetres.addChild ( maFenetre );
```

```
}

function mouvement ( pEvt:Event ):void

{

  // algorithme d'inertie
  pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
  pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;

}
```

Puis nous intégrons la logique nécessaire au sein des fonctions écouteurs `captureClic` et `clicRemontee` :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

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

var maFenetre:Fenetre;

// modification de la cadence de l'animation
stage.frameRate = 30;

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

{

       maFenetre = new Fenetre();

       maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
       maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10
);

       maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

       conteneurFenetres.addChild ( maFenetre );

}

function mouvement ( pEvt:Event ):void

{

  // algorithme d'inertie
  pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
  pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;

}

// souscription auprès du conteneur  pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, captureClic, true );

// souscription auprès du conteneur pour la phase de remontée
conteneurFenetres.addEventListener ( MouseEvent.CLICK, clicRemontee );
```

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

{

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

}

function clicRemontee ( pEvt:MouseEvent ):void

{

  var lng:int = pEvt.currentTarget.numChildren;

  var objetGraphique:DisplayObject;
  var maFenetre:Fenetre;

  while ( lng-- )

  {

       // récupération des objets graphiques
       objetGraphique = pEvt.currentTarget.getChildAt ( lng );

       // si l'un d'entre eux est de type Fenetre
       if ( objetGraphique is Fenetre )

       {

            // nous le transtypons en type Fenetre
            maFenetre = Fenetre ( objetGraphique );

            // repositionnement de chaque occurrence
            maFenetre.destX = 7 + Math.round ( lng % 3 ) * ( maFenetre.width
+ 10 );
            maFenetre.destY = 7 + Math.floor ( lng / 3 ) * ( maFenetre.height
+ 10 );

       }

  }

}
```

Nous profitons de la phase de remontée pour que le conteneur réorganise ses objets enfants lorsque l'un d'entre eux a été supprimé de la liste d'affichage.

Nous verrons au cours du chapitre 8 intitulé *Programmation orientée objet* comment notre code actuel pourrait être encore amélioré.

Ecouter plusieurs phases

Afin de suivre la propagation d'un événement, nous pouvons souscrire un écouteur auprès de chacune des phases. En affichant les valeurs des propriétés `eventPhase`, `target` et `currentTarget` de l'objet événementiel nous obtenons le chemin parcouru par l'événement.

Dans le code suivant, la fonction `ecoutePhase` est souscrite auprès des trois phases de l'événement `MouseEvent.CLICK` :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

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

var maFenetre:Fenetre;

// modification de la cadence de l'animation
stage.frameRate = 30;

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

{

       maFenetre = new Fenetre();

       maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
       maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10
);

       // souscription auprès de chaque instance pour la phase cible
       maFenetre.addEventListener ( MouseEvent.CLICK, ecoutePhase );

       maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

       conteneurFenetres.addChild ( maFenetre );

}

function mouvement ( pEvt:Event ):void

{

  // algorithme d'inertie
  pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
  pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;

}

// souscription auprès du conteneur pour la phase de remontée
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase );

// souscription auprès du conteneur  pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase, true );

function ecoutePhase ( pEvt:MouseEvent ):void

{

  /* affiche :
  1 : Phase de capture de l'événement : click
  Objet cible : [object Fenetre]
  Objet notifié : [object Sprite]
  2 : Phase cible de l'événement : click
```

```
        Objet cible : [object Fenetre]
        Objet notifié : [object Fenetre]
        3 : Phase de remontée de l'événement : click
        Objet cible : [object Fenetre]
        Objet notifié : [object Sprite]
        */

        switch ( pEvt.eventPhase )

        {

              case EventPhase.CAPTURING_PHASE :
                    trace ( pEvt.eventPhase + " : Phase de capture de l'événement : "
+ pEvt.type );
                    break;

              case EventPhase.AT_TARGET :
                    trace ( pEvt.eventPhase + " : Phase cible de l'événement : " +
pEvt.type );
                    break;

              case EventPhase.BUBBLING_PHASE :
                    trace ( pEvt.eventPhase + " : Phase de remontée de l'événement :
" + pEvt.type );
                    break;

        }

        trace( "Objet cible : " + pEvt.target );
        trace( "Objet notifié : " + pEvt.currentTarget );

      }
```

Le panneau de sortie nous affiche l'historique de la propagation de l'événement `MouseEvent.CLICK`.

A retenir

- A l'instar de la phase de capture, la phase de remontée ne concerne que les objets parents.
- La phase de remontée permet à un objet parent d'être notifié d'un événement provenant de l'un de ses enfants.
- Un même écouteur peut être souscrit aux différentes phases d'un événement.

La propriété Event.bubbles

Pour savoir si un événement participe ou non à la phase de remontée nous pouvons utiliser la propriété `bubbles` de l'objet événementiel.

Nous pouvons mettre à jour la fonction `ecoutePhase` afin de savoir si l'événement en cours participe à la phase de remontée :

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

{
```

```
  /* affiche :
  1 : Phase de capture de l'événement : click
  Objet cible : [object Fenetre]
  Objet notifié : [object Sprite]
  Evénement participant à la phase de remontée : true
  2 : Phase cible de l'événement : click
  Objet cible : [object Fenetre]
  Objet notifié : [object Fenetre]
  Evénement participant à la phase de remontée : true
  3 : Phase de remontée de l'événement : click
  Objet cible : [object Fenetre]
  Objet notifié : [object Sprite]
  Evénement participant à la phase de remontée : true
  */

  switch ( pEvt.eventPhase )

  {

        case EventPhase.CAPTURING_PHASE :
              trace ( pEvt.eventPhase + " : Phase de capture de l'événement : "
+ pEvt.type );
              break;

        case EventPhase.AT_TARGET :
              trace ( pEvt.eventPhase + " : Phase cible de l'événement : " +
pEvt.type );
              break;

        case EventPhase.BUBBLING_PHASE :
              trace ( pEvt.eventPhase + " : Phase de remontée de l'événement :
" + pEvt.type );
              break;

  }

  trace( "Objet cible : " + pEvt.target );
  trace( "Objet notifié : " + pEvt.currentTarget );

  trace ( "Evénement participant à la phase de remontée : " + pEvt.bubbles );

}
```

Notons que tout événement se propageant vers le haut participe forcément aux phases de capture et cible.

Suppression d'écouteurs

Comme nous l'avons traité lors du chapitre 3 intitulé *Le modèle événementiel*, lorsque nous souhaitons arrêter l'écoute d'un événement nous utilisons la méthode `removeEventListener`.

Lorsque nous avons souscrit un écouteur pour une phase spécifique nous devons spécifier la même phase lors de l'appel de la méthode `removeEventListener`.

Si nous écoutons un événement pour la phase de capture :

```
objetParent.addEventListener ( MouseEvent.CLICK, clicBouton, true );
```

Pour supprimer l'écoute nous devons spécifier la phase concernée :

```
objetParent.removeEventListener ( MouseEvent.CLICK, clicBouton, true );
```

Il en est de même pour les phases de remontée et cible :

```
objetParent.removeEventListener ( MouseEvent.CLICK, clicBouton, false );

objetCible.removeEventListener ( MouseEvent.CLICK, clicBouton, false );
```

Ou bien de manière implicite :

```
objetParent.removeEventListener ( MouseEvent.CLICK, clicBouton );

objetCible.removeEventListener ( MouseEvent.CLICK, clicBouton );
```

En intégrant cela dans notre exemple précédent, nous obtenons le code suivant :

```
// nombre de fenêtres
var lng:int = 12;

// création d'un conteneur
var conteneurFenetres:Sprite = new Sprite();

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

var maFenetre:Fenetre;

// modification de la cadence de l'animation
stage.frameRate = 30;

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

{

       maFenetre = new Fenetre();

       maFenetre.destX = 7 + Math.round ( i % 3 ) * ( maFenetre.width + 10 );
       maFenetre.destY = 7 + Math.floor ( i / 3 ) * ( maFenetre.height + 10
);

       // souscription auprès de chaque instance pour la phase cible
       maFenetre.addEventListener ( MouseEvent.CLICK, ecoutePhase );

       // désinscription auprès de chaque occurrence pour la phase cible
       maFenetre.removeEventListener ( MouseEvent.CLICK, ecoutePhase );

       maFenetre.addEventListener ( Event.ENTER_FRAME, mouvement );

       conteneurFenetres.addChild ( maFenetre );

}

function mouvement ( pEvt:Event ):void

{

  // algorithme d'inertie
```

```
  pEvt.target.x -= ( pEvt.target.x - pEvt.target.destX ) * .3;
  pEvt.target.y -= ( pEvt.target.y - pEvt.target.destY ) * .3;

}

// souscription auprès du conteneur  pour la phase de capture
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase, true );

// désinscription auprès du conteneur pour la phase de capture
conteneurFenetres.removeEventListener ( MouseEvent.CLICK, ecoutePhase, true
);

// souscription auprès du conteneur pour la phase de remontée
conteneurFenetres.addEventListener ( MouseEvent.CLICK, ecoutePhase );

// désinscription auprès du conteneur pour la phase de remontée
conteneurFenetres.removeEventListener ( MouseEvent.CLICK, ecoutePhase );

function ecoutePhase ( pEvt:MouseEvent ):void

{

  /* affiche :
  1 : Phase de capture de l'événement : click
  Objet cible : [object Fenetre]
  Objet notifié : [object Sprite]
  2 : Phase cible de l'événement : click
  Objet cible : [object Fenetre]
  Objet notifié : [object Fenetre]
  3 : Phase de remontée de l'événement : click
  Objet cible : [object Fenetre]
  Objet notifié : [object Sprite]
  */

  switch ( pEvt.eventPhase )

  {

        case EventPhase.CAPTURING_PHASE :
              trace ( pEvt.eventPhase + " : Phase de capture de l'événement : "
+ pEvt.type );
              break;

        case EventPhase.AT_TARGET :
              trace ( pEvt.eventPhase + " : Phase cible de l'événement : " +
pEvt.type );
              break;

        case EventPhase.BUBBLING_PHASE :
              trace ( pEvt.eventPhase + " : Phase de remontée de l'événement :
" + pEvt.type );
              break;

  }

  trace( "Objet cible : " + pEvt.target );
  trace( "Objet notifié : " + pEvt.currentTarget );

  trace ( "Evénement participant à la phase de remontée : " + pEvt.bubbles );

}
```

A retenir

- La propriété `bubbles` d'un objet événementiel permet de savoir si l'événement en cours participe à la phase de remontée.
- Lorsque nous avons souscrit un écouteur pour une phase spécifique nous devons spécifier la même phase lors de l'appel de la méthode `removeEventListener`.

Nous avons abordé au cours des chapitres précédents un ensemble de concepts clés que nous allons pouvoir mettre à profit dès maintenant.

En route pour le nouveau chapitre intitulé *Interactivité !*