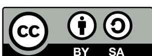
Desarrollo de aplicaciones Rich Media en la Plataforma Flash

Daniel de Fuenmayor López

PID_00192301

UOC

Universitat Oberta de Catalunya



Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-Compartir igual (BY-SA) v.3.0 España de Creative Commons. Se puede modificar la obra, reproducirla, distribuirla o comunicarla públicamente siempre que se cite el autor y la fuente (FUOC. Fundació per a la Universitat Oberta de Catalunya), y siempre que la obra derivada quede sujeta a la misma licencia que el material original. La licencia completa se puede consultar en: http://creativecommons.org/licenses/by-sa/3.0/es/legalcode.ca

Índice

1. *Integrated Development Environment* (IDE) y entornos de desarrollo 5

2. Lenguaje de programación Actionscript 7
- 2.1. Conceptos básicos 7
- 2.2. Estructuras básicas de programación 7
- 2.3. Migrando de AS2 a AS3 9
- 2.4. Programación Orientada a Objetos 12
  - 2.4.1. Conceptos básicos 12
  - 2.4.2. Clases, objetos y paquetes 13
  - 2.4.3. Propiedades y métodos 17
  - 2.4.4. Modificadores de acceso 17
  - 2.4.5. Herencia y polimorfismo 19
  - 2.4.6. Asociación de clases a Sprites visuales 20
- 2.5. Display List 21
  - 2.5.1. Conceptos básicos 21
  - 2.5.2. Trabajo dinámico con elementos 23
- 2.6. Modelo de Eventos 26
  - 2.6.1. Conceptos básicos 26
  - 2.6.2. Gestión de los eventos en AS3 27
  - 2.6.3. Eventos comunes 28
  - 2.6.4. Eventos personalizados 30
  - 2.6.5. Aplicaciones adaptadas a múltiples pantallas 31
- 2.7. Trabajo con datos externos 32
  - 2.7.1. Métodos de carga de datos 33
  - 2.7.2. Formatos de intercambio de datos 33
  - 2.7.3. Trabajo con XML y JSON 34
  - 2.7.4. *Preloaders* 40
  - 2.7.5. Imágenes 42
  - 2.7.6. Audio 44
  - 2.7.7. Vídeo 47
- 2.8. Enriqueciendo nuestras aplicaciones 49
  - 2.8.1. Mecanismos visuales: animación en línea de tiempo 49
  - 2.8.2. Animación por programación: tweens 51
  - 2.8.3. Iniciación a TweenMax 52
  - 2.8.4. Almacenaje de datos persistente: *shared objects* 54

1. *Integrated Development Environment* (IDE) y entornos de desarrollo

1) Flash Professional

En sus primeras versiones, Flash –como herramienta– focalizó esencialmente su atención en los diseñadores web. Flash no era un entorno de programación sino una herramienta multimedia. Con ella, se podía obtener Rich Media con una muy limitada interactividad (programación básica). Actionscript era, pues, en sus inicios, un lenguaje simple, de manera que la integración de diseño y programación en un mismo entorno era posible.

A lo largo de los años, Adobe ha mantenido ese propósito y Flash Professional engloba tanto una herramienta para diseñadores como una herramienta para programadores, pudiéndose realizar proyectos enteros directamente en Flash Professional. Flash Professional permite programar directamente, creando ficheros .as, editándolos, compilándolos, etc., y al mismo tiempo crear el contenido gráfico.

Pero, a pesar de ello, con la evolución sucesiva de Flash, la necesidad de poder separar la programación en una herramienta separada de Flash Professional se impone. No solo las herramientas multimedia dentro de Flash (multimedia, 3D, uso de esqueletos para animar, etc.) se multiplican, sino que Actionscript, evolucionando a AS2 y posteriormente a AS3, se ha convertido en un verdadero lenguaje de programación. Es necesario, pues, un entorno focalizado en la programación, un entorno de trabajo más próximo a Eclipse, con herramientas útiles para programadores (autocorrección, documentación, trabajo con repositorios, etc.).

Poco a poco, en la creación de ARM en Flash se ha creado una clara diferencia entre diseñadores flash (usuarios de Flash Professional o nuevas herramientas como Flash Catalyst) y programadores Actionscript, los dos pudiendo trabajar en paralelo pero en entornos dedicados diferentes.

2) Flash Builder

Evolución de su predecesor Flex Builder, Flash Builder se basa en la estructura Eclipse[1] para facilitar la programación de aplicaciones Flex o Flash. Existe como versión *standalone* (IDE independiente) o como *plugin* que podemos integrar a la IDE de Eclipse.

(1)Eclipse es quizás una de las plataformas más populares en el desarrollo software. Con versiones para los diferentes sistemas operativos, ofrece una plataforma *opensource* extensible, permitiendo a través de la instalación de diferentes *plugins* trabajar en multitud de lenguajes de programación.

Amén de ser una interfaz muy familiar para cualquier programador que trabaje ya con Eclipse (finalmente se basa en este último), una de sus mayores ventajas es su facilidad de trabajo en paralelo con Flash Professional. Se puede

programar en Flash Builder al mismo tiempo que preparamos nuestros *assets* en Flash Professional. En este aspecto, Adobe ha mejorado mucho en sus últimas versiones y el pasar de un programa a otro se realiza de forma prácticamente transparente.

Pero si de algo no se ha librado nunca Adobe es de ser un entorno cerrado y de pago. De modo que existen otras alternativas, algunas *opensource*, que pueden ser muy interesantes.

3) Otras alternativas

Flash Develop (http://www.flashdevelop.org/). Quizás uno de los entornos más conocidos. Un entorno *open source* muy completo y, sobre todo, gratuito.

FDT5 (http://fdt.powerflasher.com/). Aun no siendo gratuito, ofrece una alternativa muy completa a Flash Builder.

Enlaces relacionados

Trabajo con Flash Pro y Flash Builder:

http://help.adobe.com/es_ES/flash/cs/using/WSFD77A256-0DE1-46c7-86FB-CC4A8AE2EAA6.html.

Using Flash Builder 4 with Flash CS5:

http://tv.adobe.com/watch/flash-camp-san-francisco/using-flash-builder-4-with-flash-cs5

2. Lenguaje de programación Actionscript

2.1. Conceptos básicos

Actionscript 3 es el lenguaje de programación orientado a objetos para la plataforma Flash. Es un lenguaje basado en el estándar ECMAScript, de manera que muchas estructuras de programación resultarán familiares no solo para programadores de otros lenguajes como Javascript (basado en ECMASCript), sino también de Java o C, en los que ECMAScript se inspira.

2.2. Estructuras básicas de programación

1) Declaración de variables:

```
var nombreVariable : tipoDeVariable;
```

2) Inicialización de variables:

```
nombreVariable : tipoDeVariable = new tipoDeVariable();
nombreVariable = valor;
```

Por regla general (dentro de una clase), las instrucciones AS3 se ejecutarán de forma secuencial, una después de la otra. Pero este orden puede ser modificado de manera a romper esta secuencia, lo que comúnmente se llama transferencia de control.

Veamos rápidamente estas diferentes estructuras en AS3.

3) Estructuras selectivas:

a) Simples:

```
if (condición) instrucción;
```

b) Dobles:

```
if (condición1) {
   instrucción1;
   }
else {
   instrucción2;
   }
```

c) Compuestas (anidadas):

```
if (condición1) {
   instrucción1;
   }
else {
   if (condición2)
      instrucción2;
      }
      else {
      instrucción3
      }
   }
```

d) Múltiples:

```
switch (variable)
{
case valor1 : instrucción1;
              break;
case valor2 : instrucción2;
              break;
case valor3 : instrucción3;
              break;
default :     instrucciónDefault;
              break;
}
```

4) Estructuras repetitivas o reiterativas:

Sea como bucles independientes (cada bucle empieza y acaba independientemente) o como bucles anidados (como podría darse el caso en la lectura de una tabla de datos: un bucle se ocupa de ir de fila en fila, otro bucle se ocuparía de leer los datos de columna en columna).

a) Estructura Desde/Para:

```
for (variable, condición, expresión ) {
   // Bloque de instrucciones
   }
```

b) Estructura Mientras:

```
while (condición ) {
   // Bloque de instrucciones
   }
```

c) Estructura Repetir mientras:

```
do {
   // Bloque de instrucciones
   }
while (condición )
```

A diferencia de la estructura anterior, este bucle se ejecutará al menos una primera vez (la condición se sitúa al final)

5) Operadores booleanos:

```
OR o ||
AND o & &
NOT o !
```

Lectura recomendada

Colin Moock (2007). “Conditionals and Loops”. En: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

2.3. Migrando de AS2 a AS3

Para aquellos de vosotros que empezáis con Flash, os aconsejaríamos no entreteneros con AS2 y comenzar directamente con AS3, que no es más difícil que con AS2. Además, AS3 presenta una serie de cambios importantes, por lo que merece la pena empezar directamente por ahí.

Advertencia

Este apartado solo tiene cierto interés para aquellos de vosotros que conozcáis ya AS2.

Para aquellos programadores AS2 que hacen la transición a AS3, cabe señalar que, aunque AS2 ya supuso grandes y buenos cambios, hay que subrayar que **AS3** supone un antes y un después en el lenguaje, una estructura mucho **más orientada a POO**, **mejor organizada** y, sobre todo, **mejor optimizada**.

Desgraciadamente, esto implica un cambio en la manera de programar y nuevos conceptos a tener en cuenta. Así, como programadores AS2, a menudo os parecerá que ciertas acciones se realizan en un mayor número de pasos o que hay nuevos conceptos que os obligan a romper con ciertas costumbres bien implantadas en AS2. Es cierto en algunos aspectos, pero todo ello responde a una mejor lógica y a largo (y corto) plazo veréis que el cambio vale la pena y que os simplifica mucho la programación de aplicaciones más complejas.

Nuestro consejo, para quien conozca AS2, sería hacer borrón y cuenta nueva, empezar de cero. Teniendo en cuenta lo mucho que se mantiene (no es trabajo perdido), hay conceptos que sí cambian radicalmente.

Aquí tenéis algunos de estos puntos clave del paso de AS2 a AS3 que siempre es bueno saber.

1) Un código mejor organizado

Ya no existe la posibilidad de añadir código directamente en un símbolo (en Flash Professional haciendo clic izquierdo sobre un objeto para seleccionarlo + F9, podíamos insertarle código al símbolo).

Ahora solo podemos tener código en un *frame* o en un fichero externo .as.

Este cambio, que podría considerarse como una limitación, no hace más que simplificar la programación en AS. Uno de los mayores problemas de Flash era (es) el poder incrustar código en varios puntos de una aplicación, haciendo que el retomar un proyecto empezado pueda convertirse, rápidamente, en un juego de búsqueda del código para saber quién ejecuta qué.

Así, amén de esta mejora que limita la multiplicidad de bloques de código diseminados por la aplicación, añadiría una serie de buenas prácticas, convenciones, a la hora de programar en Flash:

- Salvo raras excepciones (acciones relativas a la cabeza de lectura en la *timeline*, como podría ser un stop() para parar una animación) hay que favorecer siempre el código externalizado. Es decir, creación de las clases de nuestra aplicación en sus respectivos ficheros .as.

- En caso de añadir código en la *timeline*, crear siempre una capa llamada Actionscrip (o AS) donde no pondremos otro contenido que el código. Por otra parte, intentar agruparlo en un solo *frame*.

De esta manera, todo el código que exista directamente en nuestro .fla será fácilmente localizable.

2) La Display List y el funcionamiento con Display Objects

Un nuevo concepto en AS3, el concepto de Display List para los elementos visibles de la aplicación. Resumiendo, la idea es que para que un objeto creado mediante programación sea visible, habrá que añadirlo a la Display List.

Ved también

Trataremos este tema en profundidad en el apartado "Display List".

Desaparecen el concepto de enlazar a objetos de la librería para luego adjuntarlos (attach) a la aplicación. Desaparece así el uso de createemptymovieclip(), duplicateMovieClip() y attachMovieClip.

En AS3, siguiendo una programación orientada a objetos, crearemos una instancia de la clase DisplayObject (en el caso de objetos en la librería Flash Professional podemos asignar una clase a un objeto de la librería utilizando el botón derecho sobre el símbolo en la librería). Luego, si queremos que esta instancia sea visible, la añadiremos a la Display List de la aplicación.

```
var balon1:Balon = new Balon(); // Crea la instancia
addChild(balon1); // La añade a la display list
```

3) Aparición de nuevos Display Objects

Obteniendo objetos más especializados se mejora el rendimiento de las aplicaciones:

- **Sprite**: Parecido a MovieClip pero sin línea de tiempos.
- **Shape**: Como MovieClip pero sin línea de tiempos ni interactividad.
- **Bitmap**: Para representar imágenes *bitmap*.
- **Loader**: DisplayObject para cargar contenido (imagenes o swf).
- **SimpleButton**: Sucesor del Button de AS2.

4) Disposición de los objetos en la Display List

Los objetos siguen disponiéndose en capas uno sobre el otro, pero desaparece el concepto de _level. Ya no existe la posibilidad de capas sin elementos. Para cambiar de posición haremos uso de swapDepth, intercambiando dos objetos pero sin por ello cambiar el número de capas de profundidad posible.

Ved también

Véase el apartado "Trabajo dinámico con elementos".

5) Propiedades en los objetos

Salvo algún cambio de sintaxis (xscale pasa a ser scaleX), la mayoría de propiedades de los Display Objects se mantienen. En cambio, se reemplaza el *underscore* por solo el punto, nomenclatura mucho más lógica y acorde con otros lenguajes de POO.

Así:

```
balon1._x
```

pasa a ser

```
balon1.x
```

Un detalle que no hay que obviar es que, en propiedades como scaleX, scaleY y alpha, los valores ya no van de 0 a 100, sino de 0 a 1; lo cual finalmente es más lógico, ya que vamos de 0%=0 a 100%=100/100=1.

6) Eventos

Se incorpora un verdadero sistema de creación y gestión de eventos.

Ved también

Véase el apartado "Modelo de eventos".

7) Clase del documento

Nueva clase del documento, clase de nuestra aplicación. Ya no es necesario tener el código en un primer *frame* de la *timeline*, sino que lo añadiremos al constructor de la clase de la aplicación.

Enlaces relacionados

Migración AS2 a AS3:

http://livedocs.adobe.com/flash/9.0_es/ActionScriptLangRefV3/migration.html.

Migración AS2 a AS3, guía de Adobe:

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/index.html.

AS3 Migration Cheatsheet:

http://actionscriptcheatsheet.com/downloads/as3cs_migration.pdf.

Introducción a AS3:

Otros enlaces de interés

http://www.mandalatv.net/fcny/.

http://actionscriptcheatsheet.com/blog/quick-referencecheatsheet-for-actionscript-20/.

Grant Skinner (2007). "Introductory AS3" (http://www.gskinner.com/talks/as3workshop/).

Dan Carr (2011). "Migrating from ActionScript 2 to ActionScript 3: Key concepts and changes" (http://www.adobe.com/devnet/flash/articles/first_as3_application.html).

2.4. Programación Orientada a Objetos

2.4.1. Conceptos básicos

La Programación Orientada a Objetos (POO) es un paradigma de programación. Dicho de otra forma, es una manera de organizarse a la hora de programar, una manera de diseñar aplicaciones. En POO, un programa está formado por una serie de objetos que interactúan entre ellos. Cada objeto por separado tendrá un conjunto de atributos (características) y métodos (comportamiento) que vienen definidos de forma común por su clase.

Ved también

La programación orientada a objetos se ve más extensamente en la asignatura *Programación web*. Puede ser interesante repasar los conceptos vistos en dicha asignatura.

Las **principales ventajas** de la POO son:

- **Relación con el mundo real**. Cualquier situación real, aplicación que queramos realizar, puede modelizarse fácilmente en POO, es decir, como un conjunto de objetos con sus características y comportamientos que interactúan entre sí.

- **Creación de sistemas más complejos pero fáciles de mantener**. En POO simplificamos un problema complejo dividiéndolo en una serie de problemas menores. Una aplicación estará compuesta por una serie de objetos que podemos tratar de forma separada. Estos objetos presentarán una menor dificultad, siendo también más fáciles de mantener.

- **Facilitar el trabajo en equipo**. Esta separación en objetos independientes permite, también, que diferentes miembros del equipo trabajen independientemente en diferentes partes de la aplicación sin entrar en conflicto.

- **Fomentar la reutilización y mejora del código.** La propia naturaleza de los objetos, al funcionar como "cajas negras", hace que puedan reutilizarse en otras aplicaciones. Es más, pueden mejorarse, obteniendo objetos que tengan las mismas características y funciones, pero que internamente estén mejor optimizados.

Algunos **conceptos importantes** asociados a la POO son:

1) Abstracción

Según la Wikipedia:

> "La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan".

En el caso de POO, la abstracción nos permite definir las características esenciales de un objeto, qué atributos y métodos necesitará. Como veremos más adelante, en POO haremos uso de las clases, que nos permitirán representar y gestionar estas abstracciones.

En otro nivel de abstracción nos ocuparemos de simplificar (dividir) una aplicación separándola en diferentes objetos (con sus funciones y atributos) y ver cómo interactúan entre ellos. A la hora de construir la aplicación, la abstracción nos permite concentrarnos en el "¿qué hacen?" y no el "¿cómo lo hacen?".

2) Encapsulamiento y ocultamiento

En POO, cada clase tiene sus propios atributos y métodos. Llamaremos **encapsulamiento** a esta facultad que tienen las clases de agrupar sus características y comportamiento.

Al mismo tiempo, una clase funcionará como una "caja negra". Una clase presentará una serie de métodos o atributos públicos, dándonos la posibilidad de controlar su comportamiento o modificar su estado desde el exterior, pero al mismo tiempo ocultará su funcionamiento interno. Esta facilidad de **ocultamiento** de su funcionamiento interno ofrece muchas ventajas desde el punto de vista de la seguridad, pero sobre todo nos permitirá el nivel de abstracción para poder construir aplicaciones más complejas. De nuevo, podemos concentrarnos en el "¿qué hacen?" y no en el "¿cómo lo hacen?".

Ved también

Veremos cómo conseguir este control de ocultamiento de atributos o métodos en el apartado "Modificadores de acceso".

Enlaces relacionados

Peter Elst (2007). "Object-oriented programming with ActionScript 3.0".

H. Paul Robertson (2010). "Creating a simple ActionScript 3 class".

Using document class:
http://www.gotoandlearn.com/play.php?id=43

2.4.2. Clases, objetos y paquetes

En POO, estructuraremos una aplicación en una serie de objetos que interactúan entre sí. Cada objeto será una instancia de una clase. Finalmente, agruparemos las clases con una misma función en paquetes.

Clases

> Una clase es un modelo (una plantilla) que sirve para crear objetos. Comúnmente, diremos que un objeto es una instancia de una clase.

Una **clase viene definida por**:

- **Sus atributos**: variables de la clase (características).

- **Sus métodos**: funciones que tiene (comportamiento).

Veamos un ejemplo, la **clase Coche**. Aunque existan coches (objetos) de todo tipo, podemos decir que todos ellos tienen características y comportamientos comunes. De esta forma podemos crear una clase Coche, plantilla que representaría los coches en general.

Esta clase Coche tendrá ciertas características, como los atributos color, número de puertas, tipo de combustible, etc., y ciertos comportamientos, como los métodos poner en marcha, apagar motor, acelerar, frenar, etc.

Otros ejemplos de clases

http://upload.wikimedia.org/wikipedia/commons/6/6d/Diagrama_de_Clases.png.

Convención de nomenclatura para las clases

- Los nombres de las clases empiezan siempre por mayúscula.
- Una instancia de una clase (objeto) empezará siempre por minúscula (ved el ejemplo anterior).

Objetos

Como hemos visto, un objeto es una entidad única que presenta unas características y comportamientos dados.

Por norma general, un objeto siempre pertenecerá a una clase de objetos. Diremos que un **objeto es una instancia de esta clase**. Este objeto es independiente de cualquier otro objeto. Recuperará los atributos y métodos de su clase, pero los valores que un objeto tome lo harán único en relación con otros objetos.

Continuando con el ejemplo anterior, y a partir de la clase Coche, podríamos crear un objeto: **miCoche**. Este objeto mantiene los métodos de la clase (acelerar, frenar, etc.) y sus atributos, pero podría tener unos valores dados. Por ejemplo, miCoche podría ser blanco, cuatro puertas, gasolina...

Diremos que el objeto miCoche es una instancia de Coche. Recuperará las características y los comportamientos de la clase Coche, pero será independiente de otra instancia de la clase Coche.

Para crear este objeto en AS3:

```
// Creamos la nueva instancia
var miCoche:Coche = new Coche();
// Asignamos los valores para los atributos de este nuevo objeto
miCoche.color = "blanco";
miCoche.numPuertas = 4;
miCoche.fuel = "gasolina";
```

Paquetes

Un paquete (*package*) agrupa una serie de clases que tienen un mismo propósito en una aplicación.

Por ejemplo, podríamos agrupar en un mismo paquete que llamaríamos *graphics* todas las clases que tengan relación con gráficos (Circulo, Rectangulo, etc.), o bien un paquete interface podría agrupar las clases relacionadas con la interfaz de nuestra aplicación (Boton, Slider, MenuDesplegable, etc.).

El hecho de **agrupar las clases en *packages* nos permitirá**:

- **Una mejor organización**. Organizar nuestras clases por sus diferentes funciones nos facilitará una mejor reutilización en otros proyectos.

- **Un mejor control de acceso (privacidad)**. Como veremos más adelante, podemos determinar que solo clases de un mismo paquete tengan acceso entre ellas. Al agruparlas en paquetes, podemos aislarlas del exterior al mismo tiempo que pueden interactuar entre ellas.

- **No entrar en conflicto con clases que tengan el mismo nombre**. Una clase viene identificada no solo por su nombre, sino por el paquete al que pertenece. El uso de paquetes permitirá que clases con un mismo nombre pero pertenecientes a paquetes diferentes no entren en conflicto.

Convención de nomenclatura para los paquetes

Como hemos visto, una de las ventajas en POO es poder aprovechar clases existentes para crear nuevas aplicaciones. Podemos no solo reutilizar nuestras propias clases, sino clases existentes en Flash (paquetes por defecto) o clases de terceros.

El hecho de poder diferenciar una clase no solo por su nombre sino por el paquete a la que pertenece permite que no haya conflictos entre clases. Pero, para ello, el nombre del paquete no solo debería incluir su finalidad (como hemos comentado), sino el nombre del proyecto y su autor. De esta forma, conseguimos nombres únicos y que no ocurran posibles conflictos con otras clases.

Este nombre no puede ser arbitrario si queremos poder colaborar con más programadores, de manera que existen una serie de convenciones a la hora de nombrar los paquetes:

- Los nombres de paquetes se escriben siempre en minúsculas para evitar conflictos con los nombres de las clases o interfaces.

- Las empresas (o autores) utilizan a menudo su nombre de dominio de Internet invertido para comenzar los nombres de sus paquetes.

Por ejemplo, el nombre de paquete com.miDominio.miProyecto.miPaquete correspondería al proyecto miProyecto creado por un programador con la DNS miDominio.com.

El uso de una DNS[2] (y no el nombre y apellido del autor) para formar el nombre del paquete nos permite obtener un nombre único a escala mundial. Pueden existir dos personas que tengan el mismo nombre e incluso los mismos apellidos, pero no habrá dos DNS iguales. De esta forma, nos aseguramos de que clases que puedan tener el mismo nombre coexistan sin entrar en conflicto, dado que el paquete es único. El hecho de invertir la DNS nos permite, simplemente, una mejor organización y reduce la posibilidad de conflicto de nombres incluso cuando se usan paquetes de terceros.

También puede ocurrir que la dirección DNS tenga un carácter no válido para un nombre de paquete. Bastaría remplazarlo por un '_' para resolver el problema.

Aunque estemos hablando de convenciones, no de obligaciones, sí que son altamente aconsejables. De hecho, además de usarlas en Actionscript, otros lenguajes de POO como Java siguen las mismas pautas, así que es un buen reflejo a adquirir.

(2)Dentro de una misma empresa con un mismo DNS se puede diferenciar entre autores añadiendo el departamento o grupo de trabajo. Por ejemplo pondríamos com.dominioDeLaEmpresa.departamento.proyecto.miPaquete).

Clases, nombre del paquete y estructura de ficheros en Flash

Una clase en Flash se representa físicamente como un fichero acabado en .as

La clase Coche, por ejemplo, se verá representada por un fichero **Coche.as**.

Por otra parte, en Flash, debido a los requisitos del compilador Actionscript de Adobe, este fichero .as deberá situarse físicamente según la estructura de carpetas indicada por el nombre del *package*.

Por ejemplo, si tenemos una clase Clase dentro del paquete com.uoc.aplicacion, el fichero Coche.as deberá cumplir obligatoriamente la estructura de ficheros siguiente:

```
src
   |- com
      |- uoc
         |- aplicacion
            |- Clase.as
```

Estructura de la declaración de una clase Clase.as en Actionscript3:

```
// Nombre del paquete (que corresponde a la ruta donde esté el fichero)
package com.uoc.aplicacion
{
      // Importamos las librerías requeridas
 import flash.display.Sprite;
      // Declaración de la clase
      // El nombre de la clase corresponde al nombre de su fichero .as
      // En este caso será 'Clase.as'
 public class Clase extends Sprite
 {
  // Declaración de los atributos
  // Constructor de la clase
      // Esta función se ejecutará en el momento de instanciar esta clase
      // Debe llevar el mismo nombre que la clase
  public function Clase()
  {
   // Inicialización de los atributos
  }
  // Declaración de los diferentes métodos de la clase
  private function metodo1() {
  }
 }
}
```

2.4.3. Propiedades y métodos

1) Propiedades

Las propiedades de una clase son variables que la clase utiliza para guardar información; son las características de una clase.

Estas propiedades pueden tener un uso interno o ser accesibles desde el exterior, definiendo, por ejemplo, los atributos específicos de una instancia particular de esta clase.

Nota

Habitualmente, no accederemos a propiedades directamente desde el exterior. Tendremos propiedades privadas y añadiremos métodos públicos (accesibles desde el exterior) para leer (métodos get) o escribir (métodos set) estas variables.

En AS3 escribiremos:

```
modificador_de_acceso var miVariable:tipo_de_variable = new tipo_de_variable
```

2) Métodos

Los métodos de una clase son las funciones que esta tiene; definen el comportamiento de una clase.

En AS3 escribiremos:

```
modificador_de_acceso function
miMetodo(atributos:tipo_tipo_de_atributo):resultado {
   //código de la función
}
```

2.4.4. Modificadores de acceso

Los modificadores de acceso controlan la accesibilidad a una propiedad (o método) según el punto donde estemos intentando acceder a ella, es decir, según su ámbito (*scope*).

Existen **cuatro tipos de modificadores de acceso**, cada cual más restrictivo que el anterior:

- Public.
- Internal.
- Protected.
- Private.

Esto nos permite controlar, proteger o dar acceso, según nos convenga, a las variables o métodos que nos interesen.

Los modificadores de acceso son una herramienta clave **para lograr elocultamiento en nuestras clases** (funcionamiento como "caja negra") y un método muy efectivo de **controlar la herencia**.

Para tener una idea global de las diferencias existentes entre cada modificador, veamos la tabla siguiente:

Posición de nuestro código	Public	Internal	Protected	Private
Código dentro de la clase donde se ha definido la variable (o método)	**Acceso permitido**	**Acceso permitido**	**Acceso permitido**	**Acceso permitido**
Código dentro de un descendiente de la clase donde se ha definido la variable (o método)	**Acceso permitido**	**Acceso permitido**	**Acceso permitido**	Acceso denegado
Código dentro de una clase diferente pero perteneciente al mismo *package* donde se ha definido la variable (o método)	**Acceso permitido**	**Acceso permitido**	Acceso denegado	Acceso denegado
Código fuera del *package* donde se ha definido la variable (o método)	**Acceso permitido**	Acceso denegado	Acceso denegado	Acceso denegado

Así, completando la estructura de la clase anterior tendremos:

```
package com.uoc.aplicacion
// Nombre del paquete (que corresponde a la ruta donde esté el fichero)
{
   import flash.display.Sprite;
   // Importamos las librerías requeridas

   public class Clase extends Sprite
   {
      // Declaración de los atributos
                  // Esta variable será solo accesible desde la propia clase
      private var nombreVariable : ClaseVariable;
                  // Esta variable será accesible desde fuera de la propia clase
      public var nombreVariable : ClaseVariable;
        public function Clase()
      // Constructor de la clase (debe llevar el mismo nombre que la clase)
      {
      // Inicialización de los atributos
      // Creación de una instancia de la clase
      nombreVariable : ClaseVariable = new ClaseVariable();
      // Valor inicial
      nombreVariable : ClaseVariable = valor;
   }
   // Declaración de los diferentes métodos de la clase
   private function metodo1() {
   }
```

```
  }
}
```

Convención de nomenclatura para variables

Es habitual el uso de *underscore* ('_') para empezar los nombres de variables privadas.

2.4.5. Herencia y polimorfismo

1) Herencia

En POO, la **herencia permite crear una nueva clase a partir de una clase padre ya existente**. Se dirá que esta nueva clase hereda de una clase padre o superclase.

La herencia nos permite aprovechar atributos y métodos de la superclase, y a la vez añadir otro tipo de funcionalidades y atributos propios a nuestra nueva clase. De esta forma, podemos de nuevo aprovechar código ya existente, objetos cuyo comportamiento y propiedades nos conviene (y funciona correctamente) para añadir solo lo que nos convenga.

En AS3 escribiremos:

```
public class Project1 extends Sprite {...}
```

Project1 hereda de Sprite: Esto nos da acceso directamente a una serie de propiedades y funciones de Sprite, como por ejemplo acceso a propiedades .buttonMode, .dropTarget, etc., o métodos como startDrag(), stopDrag(), etc. A su vez, la clase Sprite hereda las propiedades y métodos de su superclase DisplayObjectContainer, y así sucesivamente.

En general, cuando hacemos una llamada a una propiedad o método de una clase, se buscará si esta propiedad o método existe en la propia clase; en caso contrario, se buscará en su superclase, y así sucesivamente.

Una clase puede, a su vez, modificar, sobrescribir un método de su superclase (si este es público). Para ello escribiremos:

```
override public function example () {
   // Nuevo código de la función example
   }
```

que sobrescribirá la función example() de la superclase

2) Polimorfismo

El concepto de polimorfismo es el que un mismo método (mismo nombre del método) actúe de manera diferente dependiendo de la clase en que se ejecuta.

Imaginemos que tenemos dos clases Circulo y Rectangulo, con un método dibujarForma que dibuje la forma en pantalla. El polimorfismo permite que dibujarForma() haga su función pese a que internamente el código será muy diferente para el método dibujarForma() de la clase Circulo o el método dibujarForma() de la clase Rectangulo.

El polimorfismo en POO permite poder abstraerse de nuevo del funcionamiento interno de un objeto para preocuparnos solo de su uso. En el ejemplo, el dibujar la forma dada.

2.4.6. Asociación de clases a Sprites visuales

Flash nos permite asociar fácilmente *assets* gráficos (conjunto de elementos gráficos), creados directamente en Flash Profesional, a una clase.

Pongamos, por ejemplo, que queremos crear una ficha de un alumno. En esta, queremos mostrar información como su nombre, apellidos, email, una foto carnet, etc. Por otra parte, nos gustaría tener un cierto diseño gráfico (añadir un fondo o un logo, organizar los elementos visualmente, etc.) y cierta interactividad (añadir un botón para, por ejemplo, subir una nueva foto de perfil).

Para empezar, podríamos crear una clase Ficha con su atributos (nombre, apellido...) y métodos correspondientes (cambiarFoto(), editarNombre, etc.). Luego, para crear la parte gráfica, podríamos partir de cero, es decir, crear todo el aspecto gráfico vía código, creando campos de texto, situándolos en sus coordenadas respectivas x e y, etc.

Es una opción, y si el diseño es muy simple, podríamos optar por esta vía. Pero Flash Professional nos ofrece una gran ventaja a la hora de diseñar elementos. Gracias a Flash Profesional podemos crear un modelo gráfico de la ficha, podemos situar los campos, añadir un fondo, un logo, etc. Todo ello lo englobaremos dentro de un símbolo al que asociaremos la clase que queramos, en nuestro caso la clase Ficha.

Esta idea no es nueva y, a menudo, en otros lenguajes de programación veréis englobado en un solo *bitmap* todo el material gráfico de la aplicación. Desde el código, luego recuperaremos los trozos de *bitmap* que necesitemos y eso hará que no desperdiciemos tiempo redibujando elementos vía código (a menudo complicado y siempre necesitando más recursos que recuperar un simple *bitmap*).

Pero Flash añade una ventaja suplementaria. No solo podemos crear el aspecto gráfico, sino que la estructura del símbolo creado en Flash se mantiene. De esta forma, desde la clase Ficha podremos acceder directamente a sus diferentes elementos. Bastará poner el nombre adecuado a cada objeto en Flash Profesional para poder acceder a ellos desde la clase asociada.

Para **asociar una clase a un símbolo de la librería en Flash Profesional** basta con:

- Abrir la librería de símbolos (CTRL+L).
- Clicaremos el botón derecho encima del símbolo al que queremos asociarle la clase.
- Advanced > Export for Actionscript.
- Elegir la clase a la que queremos asociar este símbolo.

Lectura recomendada

C. Moock (2007). "Core Concepts". En: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

Nota

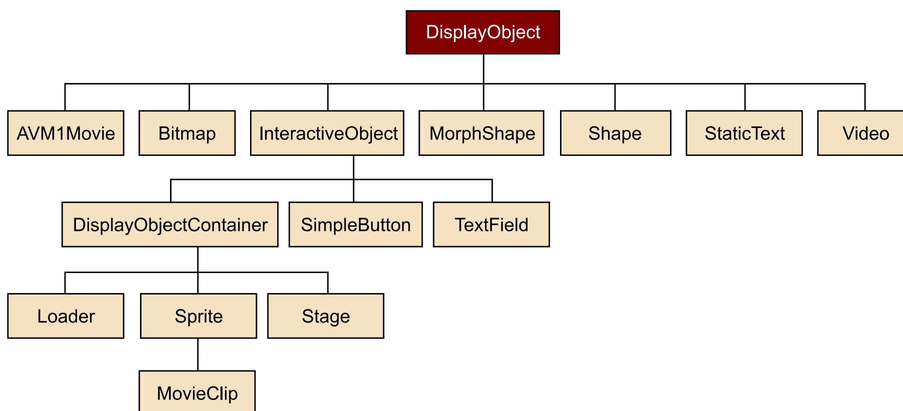
En la creación de juegos, la técnica de tener un *bitmap* que englobe los elementos de un juego (*blitting*) sigue siendo la mejor opción desde el punto de vista del rendimiento. En este caso, lo que haríamos es importar la imagen (ved el apartado "Imágenes") y recuperar los pedazos que nos interese utilizando la clase Bitmap y BitmapData.

2.5. Display List

2.5.1. Conceptos básicos

1) La clase DisplayObject

En Actionscript, todo el contenido gráfico se manipula a través de la API Display. En ella, solo un grupo de clases, descendientes de la clase DisplayObject, podrán dar lugar a objetos representables en pantalla.



Para que un **objeto sea visible**, debemos haber cumplido **dos pasos**:

- Haber creado una instancia de una de la clase DisplayObject.
- Añadir esta instancia a la display list (addChild()).

Es solo cuando añadamos este objeto a la Display List que Flash podrá mostrar su contenido en pantalla.

2) La clase DisplayObjectContainer

Como podéis ver en el árbol anterior, DisplayObjectContainer es una subclase de DisplayObject. Esta clase añade una particularidad importante, permite contener a su vez otros *display objects*.

Así, las instancias de Loader, Sprite (y por consiguiente MovieClip) y Stage pueden, a su vez, contener otras instancias de DisplayObject.

3) La Display List

En una aplicación Flash, la Display List contiene todos los objetos visibles de la aplicación. Podríamos decir que es el árbol, la estructura visible de la aplicación.

4) Estado inicial de la Display List de una aplicación

Toda aplicación Flash parte inicialmente de una **Display List compuesta por**:

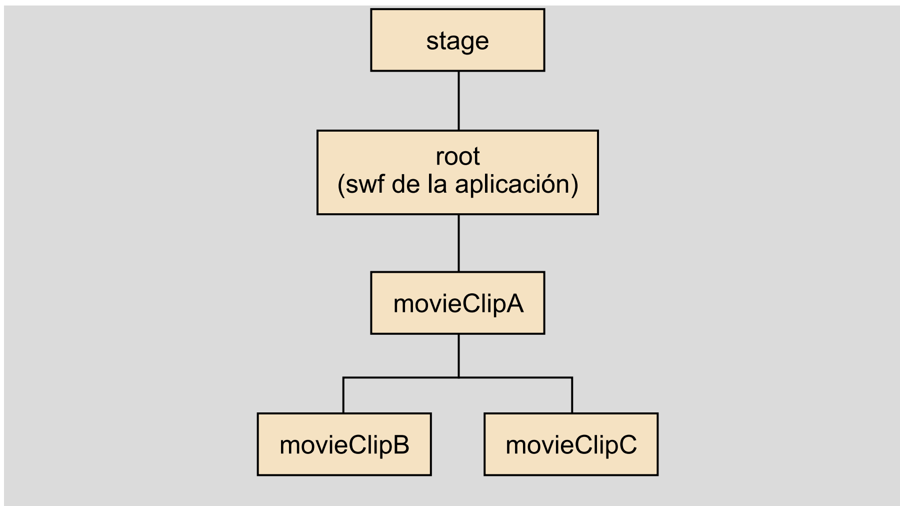
- Una instancia **stage** de la clase Stage. Este objeto es la parte superior de la jerarquía de la DisplayList.

- Hijo de este stage, tendréis **root**, una instancia de la clase principal del archivo swf, raíz de la aplicación.

- En caso de tener objetos ya creados en la *timeline* del documento .fla, estos serán descendientes directos de root.

Pongamos que, inicialmente, tenemos en una aplicación un movieClipA (instancia de MovieClip) y dentro de este tenemos dos movieclips más: movieClipA y movieClipB.

Lo que nos dará la siguiente displayList:



2.5.2. Trabajo dinámico con elementos

Para trabajar con la parte visible de la aplicación (visible en pantalla) deberemos añadir (o quitar) los objetos de esta Display List.

1) Añadir un elemento a la DisplayList

Para añadir una instancia a la DisplayList, haremos uso de **addChild**.

```
var objeto1.Sprite = new Sprite();
var objeto2:Sprite = new Sprite();
objeto1. addChild (objeto1);
```

objeto1: será el displayObjectContainer donde queremos añadir un objeto.

objeto2: será el objeto a añadir.

2) Modificar un elemento de la DisplayList

a) Opción 1: Usando el nombre de la instancia

Si estamos en el mismo ámbito donde instanciamos el objeto, podemos referirnos a él directamente.

```
var objeto1:Sprite = new Sprite();
addChild (objeto1);

// desplazamos el objeto refiriéndonos directamente al objeto
objeto1.x = 200;
```

Pero ¿qué pasa si salimos del ámbito donde se haya declarado la instancia? Ya no podemos referirnos directamente al objeto (no podremos manipularlo o eliminarlo). ¿Cómo solucionar este problema?

b) Opción 2: Uso de la propiedad .name para identificar un DisplayObjects en la Display List de su Display Container.

La clase DisplayObject tiene la propiedad .name que identifica este DisplayObject dentro de la Display List.

Por ejemplo, si dentro de cualquier ámbito hacemos:

```
var objeto1.Sprite = new Sprite();
objeto1.name="obj1";

// incluimos el objeto a la display list de la aplicación:
root.addChild(objeto1);
```

Ahora podremos recuperar el control del objeto directamente a través de la Display List. Por una parte, tenemos acceso a la Display List de root y por otro usaremos la propiedad .name para localizar el objeto que nos interesa. Entonces, haremos uso del método

`DisplayObjectContainer.getChildByName(nameChild)`

Por ejemplo, si escribimos:

```
// Gracias a la propiedad .name podemos localizar el objeto
// en la display list
root.getChildByName("obj1").x = 100;
```

3) Uso y funcionamiento de index en la DisplayList

Cuando añadimos objetos a la Display List (a un mismo Display Object Container), estos quedarán en el mismo nivel de la Display List (ved el árbol). Pero visualmente se irán superponiendo uno sobre el otro. El objeto que aparezca debajo de todos tendrá index = 0, mientras que a medida que vamos hacia el frente tendremos index = 1, 2, 3, etc.

Al igual que hacíamos con la propiedad .getChildByName, podremos recuperar el objeto que se encuentre en un índice concreto gracias al método

`DisplayObjectContainer.getChildAt(numIndex)`

```
var objeto1.Sprite = new Sprite();
var objeto2.Sprite = new Sprite();
var objeto3.Sprite = new Sprite();

// incluimos los objetos a la display list de la aplicación:
addChild(objeto1); // ---> index = 0
addChild(objeto2); // ---> index = 1
addChild(objeto3); // ---> index = 2

// si queremos mover el hijo en el índice 1, objeto2, haremos
getChildAt(1).x = 100; // moverá el objeto2 a la posición x = 100
```

Informaciones prácticas:

`getChildAt(0)`: será siempre el objeto que está más atrás.

`getChildAt (numChildren()-1)`: será el objeto más al frente.

(`numChildren` indica el número de hijos de una displayContainer.)

4) Quitar un elemento de la DisplayList

Para quitar una instancia de la displayList, podemos:

- Hacer nombreInstancia.**removeChild()** (o bien removeChild(nombreInstancia)), si se conoce el nombre de la instancia.

- Hacer **getChildByName(childName).removeChild()** (o bien removeChild(getChildByName(childName))), si se conoce su nombre.

- Hacer uso de **removeChildAt**(index), si se conoce su índice.

Atención, cuando eliminamos un objeto de la Display List, los objetos suplirán, llenarán, ese espacio vacío. Nunca queda un índice "vacío", siempre tendremos los índices cronológicamente 0,1,2..., marcando los diferentes niveles de profundidad.

```
var objeto1.Sprite = new Sprite();
var objeto2.Sprite = new Sprite();
var objeto3.Sprite = new Sprite();
var objeto4.Sprite = new Sprite();

// incluimos los objetos a la display list de la aplicación:
addChild(objeto1); // ---> index = 0
addChild(objeto2); // ---> index = 1
addChild(objeto3); // ---> index = 2
addChild(objeto4); // ---> index = 3

// si borramos el objeto2
removeChild(objeto2);

   // los índices se reorganizan para ocupar ese vacío
// objeto1 ---> index = 0
// objeto3 ---> index = 1
// objeto4 ---> index = 2
```

Otro punto muy importante a tener en cuenta es que, en ningún caso, estaremos borrando el objeto. Simplemente lo quitamos de la Display List.

Si, por ejemplo, lo quitamos de la Display List de la aplicación (Display List visible), el objeto dejará de verse en pantalla pero no dejará de existir.

5) Intercambiar la profundidad entre dos hijos

Utilizando directamente cada objeto, podemos hacer:

```
swapChildren(child1:DisplayObject, child2:DisplayObject):void
```

O podemos hacerlo utilizando los índices de cada objeto:

```
swapChildrenAt(index1:int, index2:int)
```

6) Desplazarnos dentro de la DisplayList

- `.parent`: para referirnos al objeto inmediatamente superior en la jerarquía.
- `.nombreDelHijo`: para referirnos al descendiente con el nombre "nombreDelHijo".
- `this`: para referirnos al objeto actual.

Es importante no confundir la jerarquía dentro de la DisplayList con la jerarquía existente entre clases (clases descendientes y superclases).

Lectura recomendada

Colin Moock (2007). "The Display API and the Display List". En: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

2.6. Modelo de Eventos

2.6.1. Conceptos básicos

Según la Wikipedia:

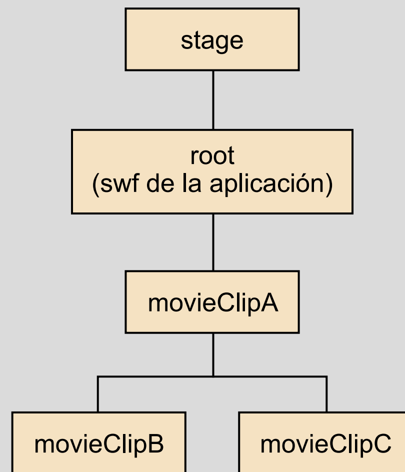
> "La programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema".

A diferencia de la programación secuencial, en la que paso a paso vamos ejecutando instrucciones, en la programación dirigida por eventos será cuando un suceso ocurra que ejecutaremos una serie de instrucciones.

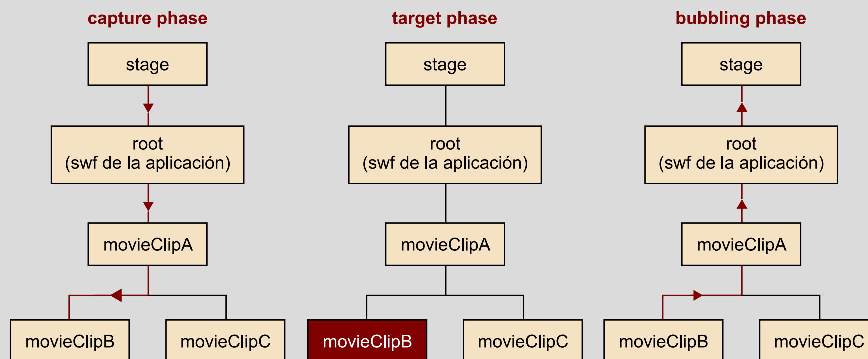
Un ejemplo de evento podría ser el haber hecho clic con el ratón o cualquier otra interacción de parte del usuario, pero también puede ser el que se haya terminado la carga de datos desde un servidor, la carga de una imagen, por ejemplo.

En AS3 cada objeto tiene sus propios eventos. Flash Player (o Air) se ocupa de gestionarlos automáticamente. De esta forma, en respuesta a un suceso creará un objeto, instancia de la clase Event (o de una clase heredera de Event), que distribuye al objeto fuente de este suceso. Si este objeto fuente forma parte de la DisplayList, el objeto evento se distribuirá siguiendo la jerarquía de la DisplayList. En algunos casos (*bubbling phase*), este objeto evento se transmitirá de vuelta por la jerarquía de la DisplayList. A este recorrido por la DisplayList se le denomina **EventFlow**.

Recordemos el ejemplo visto en el tema de la DisplayList. Esta simple aplicación contiene en la *timeline* un movieclipA y a su vez, dentro de este dos movieclips B y C. La DisplayList sería la siguiente.



Cuando hacemos clic sobre el movieclipB, Flash (o Air) distribuirán una instancia de MouseEvent que seguirá el siguiente recorrido:



Enlaces relacionados

Trevor McCauley (2008). "Introduction to event handling in ActionScript 3.0".

"Clase Event" (documentación Adobe).

Nota

Vemos que, aunque el clic se realice sobre el movieClipB, la instancia de MouseEvent recorrerá los diferentes elementos de la DisplayList (desde *stage* hasta el elemento clicado). Este recorrido del evento nos aporta una gran ventaja, permitiéndonos centralizar el código de gestión de eventos en un solo objeto. Por ejemplo, en la estructura anterior, podemos ver cómo desde root podemos detectar tanto un clic sobre movieClipA, B o C, dado que el evento MouseEvent.CLICK será siempre distribuido pasando por root (sea en la fase de captura o en la fase bubbling).

2.6.2. Gestión de los eventos en AS3

Hasta ahora, hemos visto que Flash actúa y gestiona automáticamente la distribución de eventos, dándonos información en todo momento de lo que pasa en el sistema.

Ahora, como programadores, nos interesa poder detectar estos eventos para ejecutar las instrucciones que nos interese. Desearemos escuchar un evento concreto para ejecutar una función particular.

Por ejemplo, podríamos escuchar los eventos emitidos cuando se presionen las teclas de dirección para mover nuestro personaje en un juego.

Para ello, deberemos tener en cuenta **tres datos esenciales**:

- **Qué objeto es el que estará escuchando el evento**.
- **Qué evento queremos detectar o escuchar**.
- **Qué acción queremos que se ejecute si el evento sobre este objeto ocurre**.

Advertencia

Como hemos visto, en la detección del clic sobre un elemento, no tiene por qué ser este el que escucha. Un elemento padre podrá ser más útil para centralizar la detección de los clics sobre sus elementos hijo, por ejemplo.

AS3:

```
objetoQueEscucha.addEventListener(eventoAEscuchar,funcionAEjecutar);
```

Nota

Por defecto, addEventListener escuchará solo la fase bubbling del recorrido del evento. Por ejemplo, para un evento MouseEvent.CLICK, escucharemos el recorrido desde el elemento clicado hasta la raíz, stage, de la aplicación. Pero podemos escuchar la fase de captura poniendo:

```
objetoEnQueRecaeElEvento.addEventListener(MouseEvent.CLICK,funcionAEjecutar,
true);
```

2.6.3. Eventos comunes

Por norma general, cada clase tendrá sus propios eventos, así que es muy aconsejable echar siempre un vistazo a la documentación Adobe para ver qué eventos tenemos a nuestra disposición. La lista de posibles eventos es larga y variada; veamos algunos de los más habituales.

Interacción con el usuario

1) Eventos de teclado (**KeyboardEvent**, http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/events/KeyboardEvent.html)

Son los relativos a la interacción con el teclado:

- Tecla presionada (KEY_DOWN)
- Tecla dejada de presiona (KEY_UP)

Para poder escuchar este evento, el elemento que escuche debe tener el foco, por lo que se aconseja que sea stage quién escuche estos eventos:

```
stage.addEventListener(KeyboardEvent.KEY_DOWN, moverNave);
```

Nota

Si escuchásemos este evento desde un objeto de la aplicación (y no desde stage), para detectar la interacción con el teclado, antes deberíamos darle el foco al objeto, por ejemplo haciendo clic sobre él.

2) Eventos de ratón (MouseEvent)

Eventos relacionados con la interacción con el ratón. Hay que destacar que una acción del ratón puede provocar más de un evento. Por ejemplo, el clic del ratón Flash enviará tanto un evento MOUSE_DOWN como MOUSE_UP y MOUSE_CLICK. Esto nos permite mayores posibilidades y un mayor control de las interacciones.

Entre otros, tendremos:

- El clic izquierdo con el ratón (CLICK) pero también el central (MIDDLE_CLICK) o derecho (RIGHT_CLICK).

- Doble clic del ratón (DOUBLE_CLICK).

- Detección del estado del ratón, presión del ratón (MOUSE_DOWN), dejar de presionar (MOUSE_UP). Muy útil al hacer aplicaciones en que podemos hacer clic y manteniendo apretado desplazar objetos (Drag and Drop).

- Movimiento del ratón (MOUSE_MOVE).

3) Eventos para dispositivos táctiles (TouchEvent)

Orientados a aplicaciones en Air y para pantallas táctiles (*smartphones*, *tablets*), tenemos detección de eventos táctiles como:

- TouchEvent.TOUCH_TAP
- TouchEvent.TOUCH_BEGIN
- TouchEvent.TOUCH_END
- TouchEvent.TOUCH_MOVE

Enlace relacionado

"Touch event handling". http://help.adobe.com/en_US/as3/dev/WS1ca064e08d7aa93023c59dfc1257b16a3d6-7ffe.html

O detección de gestos:

- TransformGestureEvent.GESTURE_PAN
- TransformGestureEvent.GESTURE_SWIPE
- TransformGestureEvent.GESTURE_ROTATE
- TransformGestureEvent.GESTURE_ZOOM

Nota

Para poder hacer uso de la detección de gestos, no hay que olvidar incluir antes:

```
Multitouch.inputMode = MultitouchInputMode.GESTURE;
```

Otros eventos de interés

1) Event.ENTER_FRAME

El evento Event.ENTER_FRAME se puede escuchar desde cualquier instancia de la clase DisplayObject y se emite automáticamente cada 1/velocidadDeLaAnimaciónFlash. Es decir, si nuestra aplicación corre a 30 frames/segundo, podremos escuchar el Event.ENTER_FRAME cada 1/30 segundos.

Esta característica lo hace muy útil para hacer transiciones, bucles en que incrementamos una propiedad a cada iteración (cada 1/frameRate).

Observación

De hecho, muchos paquetes de interpolaciones, *tweens* –que veremos en el apartado "Animación por programación: *tweens*"– funcionan internamente utilizando este evento.

2) TimerEvent

Para hacer uso de TimerEvent crearemos una instancia de la clase Timer. El funcionamiento es similar a Event.ENTER_FRAME, pero con un mayor grado de control dado que podemos elegir el tiempo entre eventos y el número de veces que se repetirá.

Resulta muy práctico si queremos realizar una acción a intervalos de tiempo precisos.

> Podríamos, por ejemplo, crear un reloj virtual y gracias a Timer mover la manecilla de los segundos a cada segundo (optimizando mejor los recursos así que si hiciéramos uso de Event.ENTER_FRAME).

Monitorizar la transferencia de datos

1) Event.INIT y **Event.COMPLETE**

Para, por ejemplo, controlar la inicialización de la carga de datos y el hecho de que la carga haya finalizado.

2) ProgressEvent

Para monitorizar el desarrollo de la descarga.

Ved también

Veremos con más detalle este aspecto en el apartado "*Loaders*".

2.6.4. Eventos personalizados

En la mayoría de casos, los eventos existentes son ampliamente suficientes para cualquier aplicación, pero existen momentos en los que quizás queramos crear nuestro propio evento personalizado.

Es más, un evento personalizado nos permite, si creamos otra clase heredera de Event, añadir atributos personalizados, propiedades que el propio evento transportará y que no tienen los eventos existentes.

En general, para el uso de eventos personalizados, deberemos:

- Si queremos que tenga nuevas propiedades o funciones, crearemos una nueva clase de evento descendiente de Event, con sus nuevos atributos y/o métodos.
- Tener un objeto descendiente de la clase EventDispatcher, para poder emitir eventos.

Observación

Hay que saber que la clase DisplayObject es descendiente de EventDispatcher, así que cualquier displayObject podrá por defecto emitir un evento.

AS3:

Podemos emitir un evento personalizado sin crear una nueva subclase de Event haciendo:

```
dispatchEvent(new Event("nuevoEventoPersonalizado"));
```

2.6.5. Aplicaciones adaptadas a múltiples pantallas

En la actualidad, una de las mayores dificultades al planificar el diseño de una aplicación es la multitud de dispositivos en que esta se podrá visualizar. Es más, ya no nos enfrentamos solo a diferentes dispositivos, sino que un mismo dispositivo puede tener dos modos de visualización.

Un *smartphone* o una tableta, por ejemplo, podrán verse tanto en vertical como en apaisado, lo que implica una redisposición de los elementos gráficos.

Nuestra aplicación debe poder adaptarse a estos cambios. Para ello, podemos hacer uso de dos eventos en particular:

1) Event.RESIZE

Especialmente interesante en aplicaciones de escritorio, el evento Event.RESIZE se emite cada vez que se modifican las dimensiones de la ventana del Flash Player. Para poder escuchar este evento, habrá que hacerlo desde stage:

```
stage.addEventListener(Event.RESIZE, resizeHandler);
```

Cualquier cambio del tamaño ejecutará la función resizeHandler(). Esta función, por ejemplo, podría tener en cuenta las nuevas dimensiones de la ventana para resituar los botones de forma diferente (un poco como haríamos en web con técnicas de *responsive design*).

Algunas **propiedades útiles para trabajar con el tamaño del stage** son:

- **`stage.scaleMode`**: Modo en que actúa Flash cuando se cambia el tamaño de la ventana. Por ejemplo, si ponemos
  `stage.scaleMode = StageScaleMode.NO_SCALE;`

los elementos dentro de Flash no cambiarán de tamaño al cambiar el tamaño de la ventana.

- **`stage.align`**: Alineación del stage en relación de la ventana del lector Flash. Habitualmente, lo más práctico es poner `stage.align = StageAlign.TOP_LEFT;`

- **`stage.stageWidth`**: Nos dará el ancho de nuestra ventana.

- **`stage.stageHeight`**: Nos dará la altura de nuestra ventana.

2) StageOrientationEvent

Especialmente adecuado para dispositivos móviles, AIR2 (y versiones superiores): hará que el objeto Stage emita dos tipos de evento cuando haya un cambio de orientación del dispositivo:

- **`StageOrientationEvent.ORIENTATION_CHANGING`**: Que indicará que la orientación del dispositivo está cambiando.

- **`StageOrientationEvent.ORIENTATION_CHANGE`**: Que indicará que la orientación del dispositivo ha cambiado. Las propiedades `.beforeOrientation` y `.afterOrientation` nos informarán de la orientación anterior y actual.

Lecturas recomendadas

C. Moock (2007). "Events and Display Hierarchies". En: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

C. Moock (2007). "Interactivity". En: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

2.7. Trabajo con datos externos

Hasta ahora, hemos tratado aplicaciones Rich Media con todo el contenido interno. El funcionamiento de la aplicación se realizaba con un contenido dado, incluido en las clases, fuentes .swf o fruto de cierta interacción con el usuario, pero sin intercambio de datos con el exterior. En caso de querer cambiar el contenido, cambiaríamos el código, creando una nueva aplicación.

Pero este método, que podría ser adecuado para aplicaciones simples que no vayan a cambiar nunca, no es el habitual.

> En la realidad actual, una aplicación demanda un contenido dinámico, un contenido actualizado y aplicaciones flexibles, con cierto margen de parametrización, sin por ello tener que crear cada vez una nueva aplicación.

A menudo, conectaremos con una base de datos que nos suministrará información actualizada o podremos parametrizar externamente ciertas propiedades de nuestra aplicación, todo ello sin tener que compilar de nuevo la aplicación.

2.7.1. Métodos de carga de datos

Para empezar, deberemos recuperar los datos, cargar los datos situados sea en un servidor remoto, sea localmente.

En AS3, usaremos para ello dos clases principalmente que nos facilitarán esta carga de datos:

- La clase **UrlLoader**: Especialmente orientada a la carga de datos texto o binarios, como pueden ser ficheros XML o JSON, imágenes, swfs, etc.

- La clase **Loader**: Especialmente orientada a la carga de .swf y ficheros gráficos externos.

Y seguiremos los pasos de:

- Carga **de los datos**.

- **Interpretación de los datos**, *bytes*, en un objeto que podamos manipular desde el código.

2.7.2. Formatos de intercambio de datos

A la hora de intercambiar datos, las primeras preguntas serían ¿qué formato utilizar? y ¿cómo estructurar la información y encontrar un formato común de uso para todo intercambio de datos?

> La **serialización de la información** es el proceso de codificar un objeto para poder ser guardado (en archivo o *buffer* de memoria) y enviado electrónicamente, permitiendo así el intercambio de datos.

Pero el hecho de poder tratar los datos no sería suficiente sin un formato mínimamente legible; un formato que nos permita una buena abstracción de la estructura de datos. Es decir, un formato que nos permita entender la estructura de los datos de una simple ojeada.

XML y, más recientemente, **JSON**, son dos formatos que nos permite esta serialización, obteniendo al mismo tiempo un formato legible.

Habitualmente, lo que tendremos es una base de datos en un servidor, con toda la información necesaria. Gracias a un lenguaje del lado servidor, como podría ser PHP, haciendo un *request* a la base de datos obtendremos justo la información necesaria. Esta información final, generalmente en formato XML o JSON, es la que recibirá nuestra aplicación y con la que trabajará.

2.7.3. Trabajo con XML y JSON

Trabajo con XML

El uso de XML para la serialización de datos se remonta a mucho tiempo atrás, de manera que Flash incorpora ya una serie de clases que nos permiten la lectura y manipulación de ficheros XML de manera simplificada.

Enlace relacionado

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/XML.html.

1) Estructura de un XML

La tecnología XML estructurará la información gracias a elementos anidados. Guardaremos los datos sea:

- Como contenido de un nodo.
- Como atributo de un nodo.

Y estructuraremos la información anidando los nodos adecuadamente.

Pongamos que queremos catalogar una biblioteca con una estructura XML. De cada libro tenemos un código isbn, el título del libro y el nombre del autor. Una estructura XML para este caso podría ser:

```
<?xml version="1.0" encoding="UTF-8"?>
<BIBLIOTECA>
   <LIBRO isbn = "codigo1">
      <TITULO>título 1</TITULO>
      <AUTOR>autor 1</AUTOR>
   </LIBRO>
   <LIBRO isbn = "codigo2">
      <TITULO>título 2</TITULO>
      <AUTOR>autor 2</AUTOR>
   </LIBRO>
   <LIBRO isbn="código3">
      <TITULO>título 3</TITULO>
      <AUTOR>autor 3</AUTOR>
   </LIBRO>
</BIBLIOTECA>
```

Vemos cómo para cada libro dado guardamos el código isbn como atributo de un nodo libro. Por otra parte, cada nodo libro tiene dos nodos hijo, <titulo> y <autor>, que contienen la información del título del libro y su autor, respectivamente.

La siguiente estructura XML podría también estructurar la misma información:

```
<?xml version="1.0" encoding="UTF-8"?>
<BIBLIOTECA>
<LIBRO isbn="código1" titulo="título 1" autor="autor 1"></LIBRO>
<LIBRO isbn="código2" titulo="título 2" autor="autor 2"></LIBRO>
<LIBRO isbn="código3" titulo="título 3" autor="autor 3"></LIBRO>
</BIBLIOTECA>
```

2) Lectura de datos XML

a) Recuperación del fichero XML

Lo primero que deberemos hacer es recuperar, cargar, el fichero externo. Para ello, haremos uso de la clase **URLLoader** nativa de AS3 y **URLRequest** para indicar la url donde se encuentre el archivo.

Inicializamos las variables:

```
//init variables
private var dataLoader:URLLoader;
private var url:URLRequest;
```

Cargamos los datos:

```
url:URLRequest = new URLRequest ("datos.xml");
dataLoader:URLLoader = new URLLoader ();

//detecting loading end
dataLoader.addEventListener(Event.COMPLETE, loaderCompleteHandler);

//loading data
dataLoader.load(url);
```

Nota

El fichero .xml no tiene por qué existir físicamente. De hecho, en la mayoría de los casos, haremos una llamada a un servicio web que nos dará directamente como respuesta la información en formato XML. Así, será más habitual tener algo como:

```
url:URLRequest = new URLRequest ("webService.php?parametro1= valor1")
```

Dónde webService es una función php que nos da, a partir de ciertos parámetros de búsqueda, una respuesta en formato XML.

b) Recuperación de la información (parseo del fichero XML)

Una vez hemos hecho la carga del fichero XML, obtenemos su contenido. Pero este contenido ha sido leído y recuperado como una serie de caracteres, como un String, no como una estructura de datos. Nos falta interpretar este String para poderlo organizar en un objeto y poderlo luego manipular con facilidad.

Para ello, AS3 tiene la ventaja de venir provisto de clases nativas para XML. Sabiendo que lo que estamos tratando es la estructura de un documento XML, estas clases nos permiten interpretar este String y transformarlo en un objeto estructurado.

Nos bastará crear un objeto XML a partir del String recuperado:

```
//serialized string into XML
var xml:XML;
xml = new XML(dataLoader.data);
```

c) Navegar dentro de la estructura del nuevo objeto XML

Explorar un objeto XML es muy parecido a explorar una estructura de archivos con sus consiguientes carpetas.

Partiendo de la estructura XML de la biblioteca (versión 1). Veamos algunas instrucciones de interés que podríamos utilizar:

Obtener los hijos (lista de libros que componen la biblioteca):

```
libros:XMLList = xml.children();
```

O dado que ya conocemos la estructura, podríamos escribir:

```
libros:XMLList = xml.LIBRO;
```

Lectura del valor de un atributo de un nodo (por ejemplo, autores de los libros):

```
autores:XMLList = xml.children().AUTOR;
```

O lo que es lo mismo:

```
autores:XMLList = xml.LIBRO.AUTOR;
```

Lectura del contenido de un nodo:

Utilizaremos un índice para escoger el hijo al que queremos referirnos.

Así, si queremos acceder al autor del primer libro de la biblioteca, escribiremos:

```
primerAutor:String = xml.LIBRO[0].AUTOR;
```

Nota

0 se referirá al primero de los hijos.

Lectura de un atributo de un nodo:

Para ello, haremos uso de @ para indicar el atributo que queremos leer. Siguiendo con el ejemplo de la biblioteca, si queremos leer el código isbn del primer libro haremos:

```
isbnPrimerLibro:String = xml.LIBRO[0].@isbn;
```

Exploración del objeto XML:

A menudo, no sabremos cuántos elementos componen nuestro XML. Deberemos explorar la estructura pasando de hijo a hijo para ir extrayendo la información. Varias funciones nos serán de utilidad:

- **Función length()**. Nos dará el número de nodos que compone un objeto XMLList. Así, en el ejemplo, para obtener el nombre de libros escribiremos:

```
numLibros:int = xml.LIBRO.length()
```

- **Función parent()**. Para acceder al nodo padre.

- **Función nextSibling** y **función previousSibling**. Para pasar al hermano adyacente posterior o anterior.

- **ChildIndex()**. Para conocer el índice de un nodo.

Lectura recomendada

C. Moock (2007). "XML and E4X". En: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

Trabajo con JSON

Debido, sobre todo, a una mejor representación de la información, su facilidad de procesamiento y codificación, JSON se ha convertido en poco tiempo en uno de los formatos más utilizados en transmisión de datos, reemplazando a menudo XML.

Enlaces relacionados

JSON (web oficial). http://www.json.org/

Validador JSON. http://jsonlint.com/

1) Estructura en JSON

Básicamente, tenemos dos estructuras posibles:

- Series de parejas nombre/valor. Similar a la estructura de una Associative Array en AS3.

- Listas de valores. Similar a Array en AS3.

Como vemos, JSON utiliza convenciones de escritura muy familiares a la mayoría de lenguajes de programación modernos.

En el ejemplo de la biblioteca, la versión JSON sería la siguiente:

```
{"libro": [
   {
   "isbn": "código1",
   "titulo": "título1",
   "autor": "autor1"
   },
   {
   "isbn": "código2",
   "titulo": "título2",
   "autor": "autor2"
   },
   {
   "isbn": "código3",
   "titulo": "título3",
   "autor": "autor3"
   }
   ]
}
```

Nota

Recientemente, se han añadido clases nativas directamente en Flash para la manipulación JSON: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/JSON.html.

Pero existen también librerías de terceros, como puede ser https://github.com/mikechambers/as3corelib, que han sido hasta ahora una alternativa de calidad par trabajar con JSON.

2) Pasos a seguir para la lectura de un fichero JSON utilizando as3corelib

a) Lectura del fichero

Al igual que para la lectura de datos XML, haremos uso de **URLLoader** (y su **URLRequest**) para recuperar los datos.

Inicialización de las variables:

```
//init variables
private var dataLoader:URLLoader;
private var url:URLRequest;
```

Carga de los datos:

```
url:URLRequest = new URLRequest ("datos.json");
dataLoader:URLLoader = new URLLoader ();

//detecting loading end
dataLoader.addEventListener(Event.COMPLETE, loaderCompleteHandler);

//loading data
dataLoader.load(url);
```

b) Interpretación del String obtenido y creación de un objeto estructurado con el que trabajar (parseo del JSON)

Una vez cargada la información, podremos recuperarla en dataLoader.data. Pero ¡atención!, esta información se nos presenta de nuevo como una cadena de caracteres, un String. Ahora necesitamos interpretar esta información y estructurarla en un objeto que nos pueda ser de utilidad.

Entonces, haremos uso de la librerías as3corelib, que podemos encontrar en: https://github.com/mikechambers/as3corelib/downloads, y más concretamente haremos uso del paquete com.adobe.serialization.json.

Una vez añadido el paquete a nuestra aplicación, deberemos importar la clase que nos interesa:

```
import com.adobe.serialization.json.JSON;
```

Luego, utilizando la función JSON.decode(), obtendremos el objeto estructurado a partir del String cargado:

```
var data:Object;
data = JSON.decode (dataLoader.data);
```

Nota

En la nueva clase nativa JSON de AS3, esta acción se realiza con la función JSON.parse(). (Atención: no confundir JSON nativa con JSON as3corelib).

c) Navegación y extracción de la información a partir del objeto JSON

La extracción de información sería muy parecida a la que haríamos en un sistema de Arrays. Por una parte, utilizaremos un índice entre [ ] para seleccionar un elemento dentro de un vector. Luego, con .nombreAtributo obtenemos el valor del atributo en cuestión.

Siguiendo con el ejemplo de la biblioteca (versión JSON), podríamos:

Obtener todos los libros con:

```
data.libro
```

Obtener el primer libro con:

```
data.libro[0]
```

Obtener el autor del primer libro con:

```
data.libro[0].autor
```

Obtener el número de libros de la biblioteca (equivalente a obtener el número de elementos de la Array libro) **con**:

```
data.libro.length;
```

Atención: no confundir propiedad .length con la función length() utilizada en el objeto XMLList.

2.7.4. *Preloaders*

Acabamos de ver cómo el tratamiento de datos, sea XML, JSON u otros formatos, seguiría siempre pasos parecidos:

- La petición al servidor de los datos que necesitamos.
- Espera de respuesta servidor. El servidor procesa nuestra petición.
- Se establece la conexión con el servidor y se recuperan de los datos.
- Se interpretan los datos recibidos.

Vemos que tanto para el segundo como para el tercer paso, la aplicación estará en espera, es decir, estaremos esperando una respuesta para luego descargar los datos. Estos tiempos de espera dependerán tanto de la conexión como de la respuesta del servidor o de la cantidad de información que estemos descargando, pudiendo llegar a ser considerables.

Por otra parte, el usuario, sin conocimiento de lo que esté pasando, esperará una respuesta inmediata a su acción. Viendo que esta no ocurre inmediatamente, puede pensar:

- Que no se ha detectado el clic (con lo que intentará hacer clic de nuevo, lanzando de nuevo la petición).

- Peor aún, que la aplicación no funciona y acabe por abandonarla.

Es por ello que siempre que tengamos una carga de datos o espera, habrá que indicarlo. Y de ahí la principal utilidad de los ***preloaders*** (precargadores), que es indicar al usuario que algo está ocurriendo.

Podemos diferenciar **dos tipos de *preloaders***:

a) Un ***preloader* no cuantitativo** (más habitual en la fase 2, tiempo de espera de una petición). Si no podemos estimar el tiempo de espera (como ocurre para la respuesta del servidor), utilizaremos una pequeña animación que se repita (para indicar que la aplicación no está bloqueada) y un texto explicativo que indique lo que está ocurriendo.

Por poner un ejemplo, una aplicación en que hacemos una llamada a la base de datos para recuperar información, podría tener el típico mensaje “Recuperando información de la base de datos...” con el típico *spinner* dando vueltas.

b) Un ***preloader*** **cuantitativo** (más habitual en la fase 3, fase de descarga). Siempre que tengamos conocimiento del peso final, podemos indicar con una barra de progresión (u otro elemento gráfico) el porcentaje de lo ya descargado. Este tipo de *preloader* tiene la ventaja de informar al usuario sobre el tiempo estimado que falta.

Un ejemplo muy habitual lo podemos encontrar en descargas de vídeo, donde una barra de progresión nos indica siempre que la descarga está funcionando y hasta dónde se ha descargado ya.

Una de las ventajas en AS3 es que las clases de carga de datos, tanto URLLoader o Loader, nos permiten escuchar eventos dedicados y conocer las diferentes etapas de la descarga. De esta forma, no solo podemos saber el final de una descarga (como hemos visto en la carga de datos XML o JSON con URLLoader), sino cuándo esta empieza, seguir su progresión, si hay errores, etc.

Veamos algunos de los eventos básicos para el uso de *preloaders* y la ligera diferencia de funcionamiento entre URLLoader y Loader.

1) URLLoader

En este caso, el funcionamiento es bastante directo, ya que podemos escuchar directamente desde URLoader los eventos:

- **Event.open**: Indica el inicio de la operación después de lanzar la descarga URLLoader.load().

- **Event.progress**: Recibido cuando estemos recibiendo datos de la descarga. Durante esta etapa, podremos recuperar información sobre la descarga gracias a las propiedades del propio objeto URLLoader:
  - **.bytesLoaded**: Indica el número de *bytes* descargados.

  - **.bytesTotal**: Indica el número total de *bytes* de la descarga final.

- **Event.complete**: Indica que se han descargado todos los datos.

Como hemos visto en el capítulo anterior, es en este punto donde podemos procesar los datos recibidos.

2) Loader

A pesar de que el funcionamiento es muy parecido a URLLoader, existe una diferencia a tener en cuenta.

En el caso de Loader, como con URLLoader, iniciaremos la descarga con Loader.load(), pero la escucha de eventos y recuperación de los datos no se hará directamente a través de Loader, sino de su propiedad Loader.contentLoaderInfo.

Así, será desde URLLoader.contentLoaderInfo que escuchemos:

- **Event.open**: Inicio de la descarga.

- **Event.progress**: Progresión de la descarga. En esta fase, como hemos visto anteriormente, usaremos:
  - **.bytesLoaded**: Indica el número de *bytes* descargados.

  - **.bytesTotal**: Indica el número total de *bytes* de la descarga final.

- **Event.complete**: Indica que se ha descargado toda la información.

- **Event.init**: Cuando descargamos un .swf. Nos permitirá tener acceso a propiedades del .swf final aun no habiendo acabado la descarga.

Podríamos por ejemplo, acceder a propiedades como el ancho y altura, antes de que se acabe la descarga.

Nota

El evento Init siempre ocurre antes que el evento Complete.

Hemos visto los diferentes tipos de *loaders* y sus eventos; ahora veamos su aplicación mediante ejemplos prácticos.

2.7.5. Imágenes

En la descarga de imágenes haremos uso de la clase Loader.

Veamos un ejemplo paso a paso:

1) Iniciaremos la carga (añadiendo los escuchadores de eventos que nos interese):

```
//instanciación de Loader y URLRequest
var loader:Loader = new Loader();
var urlImage:URLRequest = new URLRequest("nuestraImagen.jpg");

//Creación de escuchadores
//Ojo, observad cómo los escuchadores se aplican sobre
//loader.contentLoaderInfo
loader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS,
progressHandler);
loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
completeHandler);

//Empezamos la descarga
loader.load(urlImage);
```

2) Trataremos los eventos:

Con **ProgressEvent.PROGRESS** podremos controlar la descarga, ver cuánto se lleva descargado:

```
private function progressHandler(e:ProgressEvent):void {

   // Podemos ver los bytes cargados
   trace("Bytes cargados : " + e.bytesLoaded);

   // Ver los bytes totales
   trace("Bytes cargados : " + e.bytesTotal);

   // O hacer el cálculo del porcentaje cargado
   trace("Porcentaje cargado : " + (e.bytesLoaded/e.bytesTotal)*100
   + "%");
}
```

Es aquí donde podríamos hacer que un objeto *preloader* indicase, gracias a una barra de progresión, el porcentaje cargado.

Finalmente, con **Event.COMPLETE** detectaremos que la descarga ha finalizado y podremos procesar la imagen.

La clase Loader es una clase descendiente de DisplayObjectContainer con una particularidad importante: solo puede contener un display object en su Display List. Así, si utilizamos la misma instancia de Loader para cargar una imagen y luego una segunda imagen, la segunda imagen suprimiría la anterior.

Una manera de solucionar esto (si no queremos crear a cada carga de imagen una nueva instancia de Loader) será crear un objeto intermediario donde volcaremos la imagen recién cargada.

Lectura recomendada

C. Moock (2007). "Loading External Display Assets". En: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

Veámoslo paso a paso:

```
private function completeHandler(e:Event):void {
// La carga de la imagen ha finalizado
trace("IMAGE LOADING COMPLETED");
// Ahora tenemos acceso a las propiedades de la imagen
// Podríamos por ejemplo recuperar información sobre las dimensiones de la imagen
var imageW = e.target.width;
var stageW = stage.stageWidth;
```

```
// Supongamos que hemos creado una variable global imageContainer instancia de Sprite
// Podemos ir poniendo las imágenes cargadas dentro de la displayList de imageContainer
imageContainer.addChild(e.target.content);
// Recordad que al hacer addChild sobre otro objeto, el display object con la imagen
// desaparecerá de la display list de Loader
// De manera que podemos volver a cargar otra imagen con la misma instancia de Loader.
// Recordad que si queremos visualizar la imagen en pantalla habrá que añadir
// imageContainer a la display list de la aplicación addChild(imageContainer);
}
```

2.7.6. Audio

En la importación y manipulación de audio utilizaremos una **clase principal Sound**, que se ocupará de la descarga del sonido. Es la clase Sound la que se ocupará tanto de la descarga (*load*) y del inicio de la lectura (*play*), como del hecho de cerrar el *stream* (*close*), es decir, anular la descarga.

La manipulación del sonido se realizará a través de la clase SoundChannel. Para ello, al hacer uso del método play() se genera un nuevo objeto SoundChannel, al mismo tiempo que se lanza la lectura audio. A partir de esta nueva instancia, este objeto SoundChannel, podremos (entre otras cosas):

- Obtener información sobre el volumen actual y posición de lectura.
- Parar la lectura gracias al método stop().
- Manipular el sonido a través de un objeto SoundTransform.

Resumiendo, el tratamiento de sonido se reparte entre:

- **La clase Sound**. Permite trabajar con sonido en una aplicación. La clase Sound permite crear un objeto Sound, cargar y reproducir un archivo MP3 externo en el objeto, cerrar el flujo de sonido y acceder a datos de sonido como, por ejemplo, información sobre el número de *bytes* del flujo y los metadatos ID3.

- **La clase SoundChannel**. SoundChannel nos ayudará a controlar el sonido en una aplicación. Cada sonido está asignado a un canal de sonido y la aplicación puede tener varios canales de sonido leyéndose a la vez. La clase SoundChannel contiene un método stop(), propiedades para supervisar la amplitud (volumen) del canal y una propiedad para asignar un objeto SoundTransform en el canal.

Veamos, paso a paso, **cómo funcionaría la importación y lectura de un fichero audio externo**.

1) Descarga y monitorización (clase Sound)

Para empezar, necesitaremos instancias de Sound, URLRequest y SoundChannel:

```
//Variable declaration
private var sound1:Sound;
private var urlSound:URLRequest;
private var soundChannel1:SoundChannel;

//Instanciate variables
sound1 = new Sound();
urlSound = new URLRequest("mySong.mp3");
soundChannel1 = new SoundChannel();
```

Iniciaremos la carga del sonido, pero antes, al igual que con cualquier otra carga externa, añadiremos diferentes escuchadores de eventos que nos pueden ser útiles.

```
//Add eventListeners
   sound1.addEventListener(ProgressEvent.PROGRESS,progressHandler);
sound1.addEventListener(Event.COMPLETE,completeHandler);
sound1.addEventListener(Event.ID3,infoHandler);

//Start loading
sound1.load(urlSound);
```

El método *load* de la clase Sound tiene un funcionamiento muy similar al *load* de Loader. Nos permite, por medio del uso de eventos, monitorizar la descarga de sonido. Podríamos, por ejemplo, seguir el progreso de la descarga o si la descarga se ha completado:

```
private function progressHandler(e:ProgressEvent):void {

   trace("Loading data ... \n" + Math.floor((e.bytesLoaded/e.bytesTotal)*100) + "% charged");

}


private function completeHandler(e:Event):void {

   trace("Carga completada");

   }
```

O podríamos recuperar información ID3 de un fichero .mp3, como por ejemplo el nombre de la canción.

```
private function infoHandler(e:Event):void {
   trace("Song name : " + sound1.id3.songName);
   }
```

2) Reproducción y control de la lectura (clase Sound y SoundChannel)

La reproducción y el control de la lectura del sonido se reparte entre la clase Sound y la clase SoundChannel.

Iniciaremos la lectura del sonido gracias al método play(startTime) de la clase Sound; startTime es el tiempo en milisegundos, donde queremos que empiece la lectura.

```
soundChannel1 = sound1.play(startTIme);
```

Cada sonido en Flash se asigna a un SoundChannel, pudiendo tener diferentes objetos SoundChannel simultáneamente (sería como hablar de diferentes pistas de audio leyéndose simultáneamente). Recuperando este objeto SoundChannel, podremos controlar el sonido.

Podemos, por ejemplo, parar la lectura:

```
soundChannel1.stop();
```

O ver en qué posición de la lectura estamos, gracias a la propiedad *position*:

```
trace("Posición de lectura : "+soundChannel.position;
```

Nota

Si quisiéramos hacer un botón pausa, sería como un botón stop, pero guardaríamos en una variable la posición de lectura en la que nos encontramos.

3) Manipulación del sonido (volumen y balance con la clase SoundTransform)

Por otra parte, podremos manipular el volumen y balance del canal de sonido. Para ello, haremos uso de un objeto SoundTransform, que aplicaremos al canal de sonido (SoundChannel).

Inicialmente, creamos un objeto SoundTransform. El constructor de SoundTransform acepta unos valores iniciales de volumen y balance:

```
var soundTransform1:SoundTransform = new soundTransform (volume,pan);
//Donde volume es un valor de 0 (silencio) a 1 (volumen máximo)
//y pan un valor de -1 (balance en altavoz izquierdo) y 1 (balance
//en altavoz derecho)
```

Una vez creado el objeto, basta con aplicarlo al canal que queramos. Si queremos hacer más cambios, no hará falta crear otro objeto SoundTransform, ya que podemos modificar sus propiedades y volver a aplicarlo al canal.

Veamos, por ejemplo, cómo podríamos reducir el volumen a la mitad y centrar el balance:

```
//Modificamos el objeto SoundTransform
soundTransform1.volume = 0,5;
soundTransform1.pan = 0;

//lo aplicamos al canal
soundChannel1.soundTransform = soundTransform1;
```

2.7.7. Vídeo

Para cargar vídeos externos, haremos uso de las clases Video, NetConnection y NetStream.

El proceso se desarrolla en cuatro pasos:

1) Creación de un canal entre la aplicación y la fuente

La clase NetConnection crea una conexión bidireccional entre Flash Player (o Air) y el servidor (o emplazamiento del fichero vídeo). Se podría decir que el objeto NetConnection sirve de canal entre el cliente y el servidor.

```
private var connection:NetConnection;
connection = new NetConnection();
```

Para comprobar la conexión escucharemos los eventos NetStatusEvent.NET_STATUS y SecurityErrorEvent.SECURITY_ERROR del objeto NetConnection.

```
//Add eventlisteners
connection.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
connection.addEventListener(SecurityErrorEvent.
SECURITY_ERROR, securityErrorHandler);
```

Y utilizaremos el método connect() para crear la conexión. En caso de conexiones HTTP o si nuestro vídeo es local, haremos:

```
connection.connect(null);
```

SecurityErrorEvent.SECURITY_ERROR, como su propio nombre indica, nos informará si hay problemas de seguridad en la conexión, mientras que NetStatusEvent.NET_STATUS será quien nos informe si la conexión se ha realizado y podemos empezar la descarga.

```
private function securityErrorHandler(event:SecurityErrorEvent):void {
         trace("securityErrorHandler: " + event);
      }
private function netStatusHandler(event:NetStatusEvent):void {
         switch (event.info.code) {
            case "NetConnection.Connect.Success":
               connectStream();
               break;
            case "NetStream.Play.StreamNotFound":
               trace("Unable to locate video: " + videoURL);
            break;
         }
```

```
        }
```

2) Creación de una conexión de descarga

Una vez comprobado el canal, utilizaremos NetStream para abrir una conexión unidireccional de transmisión, que nos permitirá ir recibiendo el vídeo en *streaming* e iniciar su lectura aunque no se haya bajado en su totalidad.

```
private function connectStream():void {


   // Una vez comprobada la conexión inicializamos el objeto NetStream al que pasamos
   //el objeto NetConnection como parámetro
   stream = new NetStream(connection);

   //Al igual que con NetConnection podemos escuchar los eventos para monitorizar el
   //estado de la conexión
   stream.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
   stream.addEventListener(AsyncErrorEvent.ASYNC_ERROR, asyncErrorHandler);

   // visualizar stream de vídeo
   visualizeVideoStream();
}
```

3) Visualización del *stream* de vídeo

Ya tenemos la conexión y recepción del *stream* de vídeo. Ahora basta hacer uso de la clase Video para recuperar este *stream* y visualizarlo:

```
private function visualizeVideoStream():void {

   // Creamos una instancia de la clase Video
   video = new Video();

   // Le adjuntamos el stream vídeo
   video.attachNetStream(stream);

   // Iniciamos la lectura
   stream.play(videoURL);

   // No olvidar añadir el objeto Video a la DisplayList
   addChild(video);
}
```

4) Manipulación del *stream* de vídeo

Gracias a la clase NetStream, podemos controlar la lectura del vídeo gracias a métodos como:

```
stream.stop();          // parar lectura
stream.pause();         // pausa lectura
stream.resume();        // retoma la lectura
stream.seek(offset);    // nos permite desplazar la cabeza
                        // de lectura a la posición offset
                        // donde offset es el número
                        // de segundos desde el principio
```

O propiedades como:

```
stream.bytesLoaded;
stream.bytesTotal;   // con los que poder visualizar
                     // el porcentaje ya cargado
stream.time; // para indicarnos la posición de la cabeza
             // de lectura (en segundos)
```

Enlace relacionado

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/NetStream.html

También podemos controlar el volumen con el uso de la clase SoundTransform. El procedimiento es exactamente igual al visto con el sonido, pero lo aplicaremos a la instancia de NetStream.

Por ejemplo, para reducir el volumen del vídeo por la mitad, haríamos:

```
soundTransform1.volume = 0.5;
stream.soundTransform = soundTransform1;
```

Enlaces relacionados

Un buen ejemplo:

http://www.thetechlabs.com/audionvideo/how-to-build-a-as3-videoplayer/.

Visualizar vídeos de Youtube en una aplicación Flash gracias al API Youtube:

http://www.republicofcode.com/tutorials/flash/as3youtube/.

2.8. Enriqueciendo nuestras aplicaciones

2.8.1. Mecanismos visuales: animación en línea de tiempo

Poniendo a un lado su uso para la creación de aplicaciones Rich Media, Flash Professional se ha convertido en pocos años en una herramienta ideal para la creación de animaciones, siendo a menudo una alternativa de bajo coste para la creación de animaciones para televisión. Haciendo uso de la línea de tiempos, Flash ofrece la posibilidad de animar, mediante *keyframes* (imágenes clave), ya sea *frame* a *frame*, con interpolación de formas o de movimiento, además de otras herramientas prácticas como *onion skins*, *bones*, etc., que son muy útiles en animación.

El propósito de este capítulo no es extenderse mucho sobre este tema, que se escaparía al propósito del curso al adentrarnos más en el mundo de la animación que en el de la programación en AS3, pero sí es bueno tener una idea del funcionamiento de las interpolaciones de movimiento para luego ver las ventajas de hacerlo programáticamente.

Veamos su **funcionamiento** con un ejemplo simple:

1) Para empezar y poder aplicar una interpolación de movimiento, deberemos crear un objeto en Flash Profesional. Creamos, por ejemplo, un círculo, lo seleccionamos y creamos un objeto (F8). Desde Flash Profesional solo optaremos por gráfico o MovieClip.

2) Creamos un *keyframe* de inicio (F8) en la *timeline*, pongamos en el primer *frame*, y situamos nuestro símbolo círculo a la izquierda de la pantalla.

3) Ahora creamos otro *keyframe* (F8) más adelante en nuestra línea de tiempos (pongamos el *frame* 10) y desplazamos el círculo a la derecha de la pantalla.

4) Por último, y volviendo al primer *frame*, seleccionaremos nuestro símbolo y, clicando en el botón derecho, escogeremos interpolación clásica.

Podemos ver cómo, desplazándonos en la línea de tiempos, el círculo se mueve de la parte izquierda a la derecha.

5) La velocidad a la que se desplaza el círculo del punto A al punto B es constante (interpolación por defecto). Pero en el mundo real raramente tenemos movimientos a velocidad constante. Habitualmente, tendremos una aceleración constante, lo que nos dará una velocidad variable. Así, los objetos se aceleraran o desacelerarán de forma progresiva; raramente lo harán a una velocidad constante.

Para controlar esta evolución de la velocidad dentro de una interpolación, Flash Profesional nos proporciona la propiedad ease, que nos permitirá controlar cómo se produce la interpolación.

ease = –100, indicará una animación que empieza lenta y va acelerándose.

ease = 100, indicará una animación que empieza rápida y progresivamente va reduciendo la velocidad.

ease = 0, será una animación lineal, a velocidad constante.

Nota

También podemos tener otras formas más complejas editando la curva de variación de la velocidad directamente en Flash Profesional.

Resumiendo, podemos ver que una **interpolación de movimiento se basa en**:

- Una posición A inicial.
- Una posición B final.
- El tiempo en que pasamos de la posición A a la posición B (basado en el número de *frames* de la *timeline*).
- Escoger cómo queremos que evolucione la velocidad entre estos dos puntos, tipo de *easing*.

2.8.2. Animación por programación: tweens

Acabamos de ver cómo hacer uso de Flash Professional para obtener animaciones en la línea de tiempos. Pero este tipo de animación presenta una gran desventaja. Son animaciones fijas, estáticas. ¿Como podríamos hacer una animación interactiva?

Pongamos, por ejemplo, que queremos que la animación anterior no vaya de nuestro punto A a nuestro punto B, sino que el círculo se desplace a cualquier punto de la pantalla donde hagamos clic con el cursor del ratón. Es más, queremos que si hacemos clic en otro punto y aún no se ha acabado el primer recorrido, el círculo rectifique su trayectoria para desplazarse a este nuevo punto. ¿Cómo lo haríamos?

Vemos rápidamente que si vamos a necesitar cierta interactividad y cambios en las animaciones, hacer uso de la *timeline* será muy limitado. Sería imposible crear todas las posibles animaciones.

Es más, hasta ahora hemos hablado de interpolaciones de posición, pero ¿qué pasaría si quisiéramos cambiar otra propiedad (transparencia, tamaño, ángulo, etc.)? En este caso, se utilizan los **paquetes dedicados a crear interpolaciones (tweens)**.

Cabe destacar que, aunque AS3 tiene ya su propia clase nativa Tween, que permite crear estas interpolaciones, existen **librerías de terceros** que han demostrado, a lo largo de numerosas actualizaciones, ser mucho más interesantes y ricas, obteniendo mayor rendimiento, optimización de recursos, rapidez y mayores posibilidades de control (tipos de interpolaciones, propiedades que se pueden interpolar, uso de eventos para monitorizar la animación, etc.).

Entre las librerías más conocidas, tenemos **Tweener** (caurina), **TweenMax** (y su versión más reducida **TweenLite**), **gTween** y las que puedan ir apareciendo en un futuro.

Para el resto del curso, nos centraremos en el **paquete greensock** (TweenMax y TweenLite), un paquete muy completo. Aunque en general veréis que todas las librerías conocidas presentan características similares:

- Objeto al que aplicar la interpolación.
- Propiedad o propiedades a modificar.
- Valor/es inicial/es y final/es.
- Duración de la interpolación (en *frames* o milisegundos).
- Tipo de interpolación (*easing*).
- Eventos para monitorizar y manipular la animación.

2.8.3. Iniciación a TweenMax

Antes de empezar, deberemos descargar el paquete con las clases que necesitemos:

http://www.greensock.com/tweenmax/ y añadirlo a nuestra carpeta /com de la aplicación.

Veamos un **uso básico de TweenMax**:

1) Importamos la clase TweenMax:

```
import com.greensock.TweenMax;
```

2) Creamos una instancia de la clase TweenMax que se ocupará de la interpolación:

```
tween1 = new TweenMax (target:Object, duration:Number, vars:Object)
```

Donde:

- **target**: es el objeto al que queremos aplicar la interpolación.
- **duration**: es la duración en segundos (medida por defecto).
- **vars**: es un objeto con las propiedades a modificar.

Nota

Podemos especificar la duración en *frames*, cambiando la propiedad de la instancia tween1.useFrames = true;

Así, si queremos que el objeto *balloon* se desplace en un segundo a la posición x = 20, y = 30, y al mismo tiempo su opacidad cambie a la mitad, haríamos:

```
var tween1 = new TweenMax (balloon, 1, {
                                        x:20,
                                        y:30,
                                        alpha:0.5
                                        })
```

Cambiar el tipo de interpolación:

Para ello, añadiremos la propiedad ease al constructor, lo que se realiza a través del objeto *vars*. Atención: habrá que importar las clases para los diferentes tipos de *easing*.

```
import com.greensock.easing.*;
...
var tween1 = new TweenMax (balloon, 1, {
                                        x:20,
                                        y:30,
                                        alpha:0.5,
                                        ease:Elastic.easeOut
                                        })
```

Si no lo especificamos, TweenMax utilizará Quad.easeOut, pero existe una gran variedad de tipos de interpolación.

Detectar el final de una interpolación:

A menudo, nos interesará también detectar el final de una interpolación para ejecutar una acción particular.

Podríamos, por ejemplo, querer que un objeto cambie su opacidad progresivamente hasta ser totalmente transparente y luego borrarlo de la DisplayList (optimizando así el empleo de los recursos por la aplicación).

```
var tween1 = new TweenMax (balloon, 1, {
                                        alpha:0 ,
                                        onComplete:function()
                                              { removeChild(balloon) }
                                        })
```

A menudo, cuando queramos ejecutar más de una instrucción al finalizar una interpolación, será mejor crear una función separada que utilizaremos como *callback* al final de la interpolación. Obtendremos, así, un código mejor organizado y las instrucciones a ejecutar no quedarán condensadas dentro de la propia instanciación de TweenMax.

Retomando el ejemplo anterior, haríamos simplemente:

```
var tween1 = new TweenMax (balloon, 1, {
    alpha:0 ,
    onComplete: miMetodo
    })

// Ojo, fijaos en que solo ponemos el nombre de la función
// Sin añadir ()
// Si queremos enviar parámetros a esta función
// haremos uso del Array onCompleteParams

...

private function miMetodo():void {
//Instrucciones a realizar
removeChild(balloon);
}
```

Enlaces relacionados

Documentación:

http://www.greensock.com/as/docs/tween/_tweenmax.html.

2.8.4. Almacenaje de datos persistente: *shared objects*

En la creación de ARM, a menudo, nos encontraremos con la necesidad de conservar datos en local, directamente en el dispositivo del usuario. Podemos querer guardar las preferencias que este ha configurado, conservar el estado de la aplicación de manera que, al iniciarla de nuevo, se abra en el preciso estado en que estaba cuando se cerró, guardar datos de puntuación en un juego, etc.

Flash, a través del uso de **Shared Objects**, nos permite guardar esos datos en local. Podríamos decir que un Shared Object funcionará de forma muy similar a los *cookies* en los navegadores, con la diferencia de ser Flash quien gestiona estos ficheros.

Su funcionamiento es bastante simple.

1) Creación de una instancia de la clase Shared Object:

```
var appSharedObject:SharedObject = SharedObject.getlocal("appId");
```

Podéis ver que, a diferencia de lo acostumbrado en la creación de instancias, en este caso no haremos uso de new(). Por contra, haremos uso directamente del método getLocal(), que comprobará que el Shared Object existe:

- Si el Shared Object no existe (primera vez que lancemos la aplicación), lo creará y asignará a la instancia appSharedObject.

- Si el Shared Object ya existe (ya se ha ejecutado una primera vez la aplicación y ya lo hemos creado anteriormente), simplemente lo recuperará y asignará a appSharedObject.

2) Guardar datos en el Shared Object:

Para guardar datos en un Shared Object, primero haremos uso de la propiedad .data para crear los diferentes datos que queramos conservar. Indicaremos una variable y su valor correspondiente. Si la variable no existe, la creará, y si existe, la sobrescribirá con su nuevo valor.

```
appSharedObject.data.film => "Reservoir dogs";
appSharedObject.data.director => "Quentin Tarantino";
```

Una vez creados los diferentes datos que queramos conservar, haremos uso del método flush() para que se guarden los datos en local:

```
appSharedObject.flush();
```

3) Recuperar datos del Shared Object:

La recuperación de datos se realiza siguiendo prácticamente el mismo proceso. Haremos uso del método getLocal() para recuperar el objeto SharedObject de nuestra aplicación. Para ello, necesitaremos el id, identificador que hemos utilizado inicialmente:

```
var appSharedObject:SharedObject = SharedObject.getlocal("appId");
```

Una vez recuperado el SharedObject, bastará hacer uso de la propiedad .data para recuperar los datos que necesitemos:

```
trace (appSharedObject.data.film); &#8594; "Reservoir dogs"
trace (appSharedObject.data.director); &#8594; "Quentin Tarantino"
```

4) Borrar datos de un Shared Object:

En la creación de ARM no hay que olvidar nunca al usuario y su privacidad. Podríamos dar la posibilidad de activar/desactivar el guardar información en local o dar la posibilidad de borrar esas informaciones cuando el usuario lo desee.

Bastará hacer uso de la función clear().

```
var appSharedObject:SharedObject = SharedObject.getlocal("appId");
appSharedObject.clear();
```