Desenvolupament d'aplicacions Rich Media en la Plataforma Flash

Daniel de Fuenmayor López

PID_00192289

UOC
Universitat Oberta de Catalunya

Índex

1. *Integrated Development Environment* (IDE) i entorns de desenvolupament

1) Flash Professional

En les seves primeres versions, Flash –com a eina– va focalitzar essencialment la seva atenció en els dissenyadors web. Flash no era un entorn de programació, sinó una eina multimèdia. Amb ella es podia obtenir Rich Media amb una interactivitat molt limitada (programació bàsica). Actionscript era doncs, en els seus inicis, un llenguatge simple, de manera que era possible la integració de disseny i programació en un mateix entorn.

Al llarg dels anys Adobe ha mantingut aquest propòsit i Flash Professional engloba tant una eina per a dissenyadors com una eina per a programadors, i es poden realitzar projectes sencers directament en Flash Professional. Flash Professional permet programar directament, creant fitxers .as, editant-los, compilant-los, etc., i, alhora, creant el contingut gràfic.

Però malgrat això, amb l'evolució successiva de Flash, la necessitat de poder separar la programació en una eina independent de Flash Professional s'imposa. No solament les eines multimèdia dins de Flash (multimèdia, 3D, ús d'esquelets per a animar, etc.) es multipliquen, sinó que Actionscript evoluciona a AS2 i posteriorment a AS3 i es converteix en un veritable llenguatge de programació. Cal, doncs, un entorn focalitzat a la programació, un entorn de treball més proper a Eclipse, amb eines útils per a programadors (autocorrecció, documentació, treball amb repositoris, etc.).

A poc a poc en la creació d'ARM en Flash s'ha creat una clara diferència entre dissenyadors Flash (usuaris de Flash Professional o noves eines com ara Flash Catalyst) i programadors Actionscript, de manera que els dos poden treballar en paral·lel però en entorns dedicats diferents.

2) Flash Builder

Evolució del seu predecessor Flex Builder, Flash Builder es basa en l'estructura Eclipse[1] per a facilitar la programació d'aplicacions Flex o Flash. Existeix com a versió *standalone* (IDE independent) o com a connector que podem integrar en la IDE d'Eclipse.

A més de ser una interfície molt familiar per a qualsevol programador que treballi ja amb Eclipse (al capdavall, s'hi basa), una dels seus avantatges més destacables és la facilitat de treball en paral·lel amb Flash Professional. Es pot programar en Flash Builder al mateix temps que preparem els nostres *assets*

(1)Eclipse és potser una de les plataformes més populars en el desenvolupament de programari. Amb versions per als diferents sistemes operatius, ofereix una plataforma de font pública (*open source*) extensible, que permet, per mitjà de la instal·lació de diferents connectors, treballar en una gran diversitat de llenguatges de programació.

en Flash Professional. En aquest aspecte, Adobe ha millorat molt en les seves últimes versions i el pas d'un programa a l'altre es fa de manera pràcticament transparent.

Però si d'alguna cosa no s'ha lliurat mai Adobe és de ser un entorn tancat i de pagament. Per això existeixen altres alternatives, algunes de font pública, que poden ser molt interessants.

3) Altres alternatives

Flash Develop (http://www.flashdevelop.org/). Potser un dels entorns més coneguts. Un entorn de font pública molt complet i, sobretot, gratuït.

FDT5 (http://fdt.powerflasher.com/). Tot i que no és gratuït, ofereix una alternativa molt completa a Flash Builder.

Enllaços relacionats

Treball amb Flash Pro i 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. Llenguatge de programació ActionScript

2.1. Conceptes bàsics

Actionscript 3 és el llenguatge de programació orientat a objectes per a la plataforma Flash. És un llenguatge basat en l'estàndard ECMAScript, de manera que moltes estructures de programació resultaran familiars no solament a programadors d'altres llenguatges com Javascript (basat en ECMAScript), sinó també de Java o C, en els quals s'inspira ECMAScript.

2.2. Estructures bàsiques de programació

1) Declaració de variables:

```
var nomVariable : tipusDeVariable;
```

2) Inicialització de variables:

```
nomVariable : tipusDeVariable = new tipusDeVariable();
nomVariable = valor;
```

Per regla general (dins d'una classe) les instruccions AS3 s'executaran de forma seqüencial, una després de l'altra. Però aquest ordre es pot modificar per tal de trencar aquesta seqüència, situació que s'anomena habitualment *transferència de control*.

Vegem ràpidament aquestes diferents estructures en AS3.

3) Estructures selectives:

a) Simples:

```
if (condició) instrucció;
```

b) Dobles:

```
if (condició1) {
    instrucció1;
    }
else {
    instrucció2;
    }
```

c) Compostes (niades):

```
if (condició1) {
    instrucció1;
    }
else {
    if (condició2) {
        instrucció2;
        }
        else {
        instrucció3
        }
    }
```

d) Múltiples:

```
switch (variable)
{

case valor1 : instrucció1;
              break;

case valor2 : instrucció2;
              break;

case valor3 : instrucció3;
              break;

default     : instruccióDefault;
              break;
}
```

4) Estructures repetitives o reiteratives:

Poden aparèixer com a bucles independents (cada bucle comença i acaba independentment) o com a bucles niats (com podria ser el cas en la lectura d'una taula de dades, en què un bucle s'ocupa d'anar de fila en fila i un altre bucle s'ocupa de llegir les dades de columna en columna).

a) Estructura "des de / per a":

```
for (variable, condició, expressió ) {
    // Bloc d'instruccions
    }
```

b) Estructura "mentre":

```
while (condició) {
    // Bloc d'instruccions
    }
```

c) Estructura "repetir mentre":

```
do {
    // Bloc d'instruccions
    }
while (condició )
```

A diferència de l'estructura anterior, aquest bucle s'executarà almenys una primera vegada (la condició se situa al final).

5) Operadors booleans:

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

Lectura recomanada

Colin Moock (2007). "Conditionals and Loops". A: *Essential Actionscript 3.0*. Chambersburg: O'Reilly Media.

2.3. Migració d'AS2 a AS3

Advertiment

Aquest apartat solament té interès per a aquells de vosaltres que ja conegueu AS2.

Per a aquells de vosaltres que comenceu amb Flash, us aconsellem que no us entretingueu amb AS2 i comenceu directament amb AS3, que no és pas més difícil que AS2. A més a més, AS3 presenta una sèrie de canvis importants, per la qual cosa val la pena començar-hi directament.

Per a aquells programadors AS2 que fan la transició cap a AS3, malgrat que AS2 ja va suposar canvis importants i bons, cal subratllar que **AS3** representa un abans i un després en el llenguatge, una estructura molt més **orientada a POO**, **més ben organitzada** i sobretot **més optimitzada**.

Malauradament, això implica un canvi en la manera de programar i nous conceptes que cal tenir en compte. Així, com a programadors d'AS2, sovint us semblarà que certes accions es realitzen en més passos o que hi ha nous conceptes que us obliguen a trencar amb determinats costums ben implantats en AS2. Això és cert en alguns aspectes, però respon a una lògica millor, i a llarg (i curt) termini veureu que el canvi val la pena i que us simplifica molt la programació d'aplicacions més complexes.

El nostre consell per als que conegueu AS2, seria fer creu i ratlla, començar de zero. Cal tenir en compte tot el que es conserva, que és molt (no és treball perdut), però hi ha conceptes que sí canvien radicalment.

A continuació teniu alguns d'aquests punts clau del pas d'AS2 a AS3 que convé saber.

1) Un codi més ben organitzat

Ja no existeix la possibilitat d'afegir codi directament en un símbol (amb Flash Professional fent clic esquerre sobre un objecte per a seleccionar-lo + F9, podíem inserir codi al símbol).

Ara només podem tenir codi en un marc (*frame*) o en un fitxer extern .as.

Aquest canvi, que es podria considerar una limitació, en realitat simplifica la programació en AS. Un dels problemes més importants de Flash era (és) el fet de poder incrustar codi en diversos punts d'una aplicació, i això feia que reprendre un projecte començat es pogués convertir ràpidament en un joc de recerca del codi per a saber qui executa què.

Així, a més d'aquesta millora que limita la multiplicitat de blocs de codi disseminats per l'aplicació, cal afegir una sèrie de bones pràctiques, convencions, a l'hora de programar en Flash:

- Excepte en situacions ben poc habituals (accions relatives al capçal de lectura en la línia de temps (*timeline*), com podria ser un stop() per a aturar una animació) cal afavorir sempre el codi externalitzat. És a dir, creació de les classes de la nostra aplicació en els seus respectius fitxers .as.

- En cas d'afegir codi en la línia de temps, crear sempre una capa anomenada Actionscrip (o AS) en la qual no posarem cap altre contingut que el codi. D'altra banda, intentar agrupar-ho en un sol *frame*.

D'aquesta manera, tot el codi que existeixi directament en el nostre .fla serà fàcilment localitzable.

2) La Display List i el funcionament amb Display Objects

Es tracta d'un nou concepte en AS3, el concepte de Display List per als elements visibles de l'aplicació. Resumint, la idea és que perquè un objecte creat mitjançant programació sigui visible, caldrà afegir-lo a la Display List.

Vegeu també

Tractarem aquest tema en profunditat en l'apartat "Display List".

Desapareix el concepte d'enllaçar a objectes de la biblioteca i després adjuntar-los (*attach*) a l'aplicació. Desapareix així l'ús de createemptymovieclip(), duplicateMovieClip() i attachMovieClip.

En AS3, seguint una programació orientada a objectes, crearem una instància de la classe DisplayObject (en el cas d'objectes en la biblioteca Flash Professional podem assignar una classe a un objecte de la biblioteca utilitzant el botó dret sobre el símbol en la biblioteca). Després, si volem que aquesta instància sigui visible l'afegirem a la Display List de l'aplicació.

```
var balon1:Balon = new Balon();   // Crea la instància
addChild(balon1);                 // L'afegeix a la display list
```

3) Aparició de nous Display Objects

Si obtenim objectes més especialitzats es millora el rendiment de les aplicacions:

- **Sprite**: Semblant a MovieClip però sense línia de temps.
- **Shape**: Com MovieClip però sense línia de temps ni interactivitat.
- **Bitmap**: Per a representar imatges de mapa de bits.
- **Loader**: DisplayObject per a carregar contingut (imatges o swf).
- **SimpleButton**: Successor del Button d'AS2.

4) Disposició dels objectes en la Display List

Els objectes continuen disposant-se en capes un sobre l'altre, però desapareix el concepte de _level. Ja no existeix la possibilitat de capes sense elements. Per a canviar de posició utilitzarem swapDepth, que intercanvia dos objectes però sense canviar el nombre de capes de profunditat possible.

Vegeu també

Vegeu l'apartat "Treball dinàmic amb elements".

5) Propietats en els objectes

Excepte algun canvi de sintaxi (xscale passa a ser scaleX) la majoria de propietats dels Display Objects es mantenen; en canvi, es reemplaça la ratlla baixa pel punt, únicament, nomenclatura molt més lògica i d'acord amb altres llenguatges de POO.

Així,

```
balon1._x
```

passa a ser:

```
balon1.x
```

Un detall que no cal obviar és que en propietats com scaleX, scaleY i alpha, els valors ja no van de 0 a 100, sinó de 0 a 1; cosa que, al capdavall, és més lògica ja que anem de 0% = 0 a 100% = 100/100 = 1.

6) Esdeveniments

S'incorpora un veritable sistema de creació i gestió d'esdeveniments.

Vegeu també

Vegeu l'apartat "Model d'esdeveniments".

7) Classe del document

Nova classe del document, classe de la nostra aplicació. Ja no cal tenir el codi en un primer marc de la línia de temps, sinó que l'afegirem al constructor de la classe de l'aplicació.

Altres enllaços d'interès

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

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

Enllaços relacionats

Migració d'AS2 a AS3:

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

Migració d'AS2 a AS3, guia d'Adobe:

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

AS3 Migration Cheatsheet:

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

Introducció a AS3:

Grant Skinner (2007). "Introductory AS3".

Dan Carr (2011). "Migrating from ActionScript 2 to ActionScript 3: Key concepts and changes".

2.4. Programació orientada a objectes

2.4.1. Conceptes bàsics

La programació orientada a objectes (POO) és un paradigma de programació. Dit d'una altra forma, és una manera d'organitzar-se a l'hora de programar, una manera de dissenyar aplicacions. En POO, un programa està format per una sèrie d'objectes que interactuen entre si. Cada objecte per separat tindrà un conjunt d'atributs (característiques) i mètodes (comportament) que són definits de manera comuna per a la seva classe.

Vegeu també

La programació orientada a objectes es veu més extensament a l'assignatura *Programació web*. Pot ser interessant repassar els conceptes vistos en aquella assignatura.

Els **principals avantatges** de la POO són:

- **Relació amb el món real**. Qualsevol situació real, aplicació que vulguem realitzar, es pot modelitzar fàcilment en POO, és a dir, com un conjunt d'objectes amb les seves característiques i comportaments i que interactuen entre si.

- **Creació de sistemes més complexos però fàcils de mantenir**. En POO simplifiquem un problema complex dividint-lo en una sèrie de problemes més petits. Una aplicació estarà composta per una sèrie d'objectes que podem tractar de forma separada. Aquests objectes tindran una dificultat més baixa i també seran més fàcils de mantenir.

- **Facilitació del treball en equip**. Aquesta separació en objectes independents permet també que diferents membres de l'equip treballin independentment en diferents parts de l'aplicació sense entrar en conflicte.

- **Foment de la reutilització i millora del codi**. La naturalesa dels objectes, en funcionar com a "caixes negres", fa que es puguin reutilitzar en altres aplicacions. És més, es poden millorar i s'obtenen objectes amb les mateixes característiques i funcions, però més optimitzats internament.

Alguns **conceptes importants** associats a la POO són:

1) Abstracció

Segons la Viquipèdia:

"L'abstracció consisteix a aïllar un element del seu context o de la resta dels elements que l'acompanyen."

En el cas de la POO, l'abstracció ens permet definir les característiques essencials d'un objecte, quins atributs i mètodes necessitarà. Com veurem més endavant, en POO farem servir les classes, que ens permetran representar i gestionar aquestes abstraccions.

En un altre nivell d'abstracció ens ocuparem de simplificar (dividir) una aplicació, separant-la en diferents objectes (amb les seves funcions i atributs) i veurem com interactuen entre ells. A l'hora de construir l'aplicació, l'abstracció ens permet concentrar-nos en el "què fan?" i no en el "com ho fan?".

2) Encapsulament i ocultació

En POO cada classe té els seus atributs i mètodes propis. Anomenem **encapsulament** aquesta facultat que tenen les classes d'agrupar les seves característiques i comportament.

Igualment, una classe funcionarà com una "caixa negra". Una classe tindrà una sèrie de mètodes o atributs públics i ens oferirà la possibilitat de controlar el seu comportament o modificar el seu estat des de l'exterior, però, alhora, n'ocultarà el funcionament intern. Aquesta facilitat d'ocultació del funcionament intern ofereix molts avantatges en l'àmbit de la seguretat, però sobretot ens permetrà el grau d'abstracció necessari per a poder construir aplicacions més complexes. De nou podem concentrar-nos en el "què fan?" i no en el "com ho fan?".

Vegeu també

Veurem com aconseguir aquest control d'ocultació d'atributs o mètodes en l'apartat "Modificadors d'accés".

Enllaços relacionats

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. Classes, objectes i paquets

En POO estructurarem una aplicació en una sèrie d'objectes que interactuen entre si. Cada objecte serà una instància d'una classe. Finalment, agruparem en paquets les classes amb una mateixa funció.

Classes

Una classe és un model (una plantilla) que serveix per a crear objectes. Habitualment direm que un objecte és una instància d'una classe.

Una **classe és definida per**:

- **Els seus atributs**: variables de la classe (característiques).
- **Els seus mètodes**: funcions que té (comportament).

Vegem un exemple, la **classe Cotxe**. Encara que existeixin cotxes (objectes) de tot tipus, podem dir que tots ells tenen característiques i comportaments comuns. D'aquesta manera podem crear una classe Cotxe, plantilla que representaria els cotxes en general.

Aquesta classe Cotxe tindrà certes característiques, com els atributs color, nombre de portes, tipus de combustible, etc., i certs comportaments, com els mètodes engegar, apagar motor, accelerar, frenar, etc.

Altres exemples de classes

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

Convenció de nomenclatura per a les classes

- Els noms de les classes comencen sempre amb majúscula.
- Una instància d'una classe (objecte) començarà sempre amb minúscula (vegeu l'exemple anterior).

Objectes

Com hem vist, un objecte és una entitat única que té unes característiques i comportaments determinats.

Per norma general, un objecte sempre pertanyerà a una classe d'objectes. Direm que un **objecte és una instància d'aquesta classe**. Aquest objecte és independent de qualsevol altre objecte. Recuperarà els atributs i mètodes de la seva classe, però els valors que prengui un objecte el faran únic en relació amb altres objectes.

Continuant amb l'exemple anterior, i a partir de la classe Cotxe, podríem crear un objecte: **elmeuCotxe**. Aquest objecte manté els mètodes de la classe (accelerar, frenar, etc.) i els seus atributs, però podria tenir uns valors determinats. Per exemple, elmeuCotxe podria ser blanc, quatre portes, gasolina, etc.

Direm que l'objecte elmeuCotxe és una instància de Cotxe. Recuperarà les característiques i comportaments de la classe Cotxe però serà independent d'una altra instància de la classe Cotxe.

Per a crear aquest objecte en AS3:

```
// Creem la nova instància
var elmeuCotxe:Cotxe = new Cotxe();

// Assignem els valors per als atributs d'aquest nou objecte
elmeuCotxe.color = "blanc";
elmeuCotxe.numPortes = 4;
elmeuCotxe.fuel = "gasolina";
```

Paquets

Un paquet (*package*) agrupa una sèrie de classes que tenen un mateix propòsit en una aplicació.

Per exemple, podríem agrupar en un mateix paquet que anomenaríem *graphics* totes les classes que tinguin relació amb gràfics (Cercle, Rectangle, etc) o bé un paquet *interface* podria agrupar les classes relacionades amb la interfície de la nostra aplicació (Botó, Slider, MenuDesplegable, etc).

El fet **d'agrupar les classes en paquets ens permetrà**:

- **Una millor organització**. Organitzar les nostres classes per les seves diferents funcions ens facilitarà una reutilització més eficient en altres projectes.

- **Un millor control d'accés (privadesa)**. Com veurem més endavant, podem determinar que només classes d'un mateix paquet tinguin accés entre si. En agrupar-les en paquets, podem aïllar-les de l'exterior i, al mateix temps, poden interactuar entre si.

- **No entrar en conflicte amb classes que tinguin el mateix nom**. Una classe queda identificada no solament pel seu nom, sinó pel paquet al qual pertany. L'ús de paquets permetrà que classes amb un mateix nom però pertanyents a paquets diferents no entrin en conflicte.

Convenció de nomenclatura per als paquets

Com hem vist, un dels avantatges en POO és el fet de poder aprofitar classes existents per a crear noves aplicacions. Podem reutilitzar no solament les nostres pròpies classes, sinó classes existents en Flash (paquets per defecte) o classes de tercers.

El poder diferenciar una classe, no solament pel seu nom sinó pel paquet a la qual pertany, permet que no hi hagi conflictes entre classes. Però per a això el nom del paquet no hauria d'incloure únicament la seva finalitat (com hem comentat), sinó també el nom del projecte i el seu autor. D'aquesta manera aconseguim noms únics i que no es produeixin possibles conflictes amb altres classes.

Aquest nom no pot ser arbitrari si volem poder col·laborar amb més programadors, de manera que existeixen una sèrie de convencions a l'hora de nomenar els paquets:

- Els noms de paquets s'escriuen sempre amb minúscules, per tal d'evitar conflictes amb els noms de les classes o interfícies.

- Les empreses (o autors) utilitzen sovint el seu nom de domini d'Internet invertit per a començar els noms dels seus paquets.

Per exemple, el nom de paquet com.elmeuDomini.elmeuProjecte.elmeuPaquet correspondria al projecte elmeuProjecte creat per un programador amb la DNS elmeuDomini.com.

L'ús d'una DNS[2] (i no el nom i cognom de l'autor) per a formar el nom del paquet ens permet obtenir un nom únic a escala mundial. Poden existir dues persones que tinguin el mateix nom i, fins i tot, els mateixos cognoms, però no hi haurà dos DNS iguals. D'aquesta manera ens assegurem que classes que puguin tenir el mateix nom coexisteixin sense entrar en conflicte, atès que el paquet és únic. El fet d'invertir la DNS ens permet simplement una millor organització i redueix la possibilitat de conflicte de noms fins i tot quan s'usen paquets de tercers.

Malgrat que estiguem parlant de convencions, no d'obligacions, sí que són altament aconsellables. De fet, a més de fer-les servir en Actionscript, altres llenguatges de POO com Java segueixen les mateixes pautes, de manera que és un bon costum adquirir-les.

(2)Dins d'una mateixa empresa amb un mateix DNS es pot diferenciar entre autors afegint el departament o grup de treball. Per exemple posaríem, com.dominiDeLEmpresa.departament.projecte. elmeuPaquet).
També pot passar que l'adreça DNS tingui un caràcter no vàlid per a un nom de paquet. En aquest cas n'hi hauria prou de canviar-lo per un '_' per a resoldre el problema.

Classes, nom del paquet i estructura de fitxers en Flash

Una classe en Flash es representa físicament com un fitxer acabat en .as

La classe Cotxe, per exemple, estarà representada per un fitxer **Cotxe.as**.

D'altra banda, en Flash, a causa dels requisits del compilador Actionscript d'Adobe, aquest fitxer .as s'haurà de situar físicament segons l'estructura de carpetes indicada pel nom del paquet.

Per exemple, si tenim una classe Classe dins del paquet com.uoc.aplicacio, el fitxer Cotxe.as haurà de complir obligatòriament l'estructura de fitxers següent:

```
src
  |- com
    |- uoc
      |- aplicacio
        |- Classe.as
```

Estructura de la declaració d'una classe Classe.as en Actionscript3:

```
// Nom del paquet (que correspon a la ruta on estigui el fitxer)
package com.uoc.aplicacio

{
       // Importem les biblioteques requerides
  import Flash.display.Sprite;

       // Declaració de la classe
       // El nom de la classe correspon al nom del seu fitxer .as
       // En aquest cas serà 'Classe.as'
  public class Classe extends Sprite
  {
    // Declaració dels atributs

    // Constructor de la classe
            // Aquesta funció s'executarà en el moment d'instanciar aquesta classe
            // Ha de dur el mateix nom que la classe
    public function Classe()
        {

            // Inicialització dels atributs
    }

        // Declaració dels diferents mètodes de la classe
    private function metode1() {
        }
```

```
    }
}
```

2.4.3. Propietats i mètodes

1) Propietats

Les propietats d'una classe són variables que la classe utilitza per a desar informació; són les característiques d'una classe.

Nota

Habitualment no accedirem a propietats directament des de l'exterior. Tindrem propietats privades i afegirem mètodes públics (accessibles des de l'exterior) per a llegir (mètodes get) o escriure (mètodes set) aquestes variables.

Aquestes propietats poden tenir un ús intern o ser accessibles des de l'exterior, si es defineixen per exemple els atributs específics d'una instància particular d'aquesta classe.

En AS3 escriurem:

```
modificador_de_acces var lamevaVariable:tipus_de_ variable = new tipus_de_variable
```

2) Mètodes

Els mètodes d'una classe són les funcions que té; defineixen el comportament d'una classe.

En AS3 escriurem:

```
modificador_de_acces
function elmeuMetode(atributs:tipus_tipus_de_atribut):resultat {
  // codi de la funció
}
```

2.4.4. Modificadors d'accés

Els modificadors d'accés controlen l'accessibilitat a una propietat (o mètode) segons el punt en què estiguem intentant accedir-hi, és a dir segons el seu àmbit (*scope*).

Hi ha **quatre tipus de modificadors d'accés**, cada un més restrictiu que l'anterior:

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

Això ens permet controlar, protegir o donar accés, segons ens convingui, a les variables o mètodes que ens interessi.

Els modificadores d'accés són una eina clau **per a aconseguir l'ocultació en les nostres classes** (funcionament com a "caixa negra") i un mètode molt efectiu de **controlar l'herència**.

Per a tenir una idea global de les diferències existents entre cada modificador vegeu la taula següent:

Posició del nostre codi	Public	Internal	Protected	Private
Codi dins de la classe on s'ha definit la variable (o mètode)	**Accés permès**	**Accés permès**	**Accés permès**	**Accés permès**
Codi dins d'un descendent de la classe on s'ha definit la variable (o mètode)	**Accés permès**	**Accés permès**	**Accés permès**	Accés denegat
Codi dins d'una classe diferent però que pertany al mateix paquet en què s'ha definit la variable (o mètode)	**Accés permès**	**Accés permès**	Accés denegat	Accés denegat
Codi fora del paquet en què s'ha definit la variable (o mètode)	**Accés permès**	Accés denegat	Accés denegat	Accés denegat

Així, completant l'estructura de la classe anterior, tindrem:

```
package com.uoc.aplicacio
// Nom del paquet (que correspon a la ruta on estigui el fitxer)
{
    import Flash.display.Sprite;
    // Importem les llibreries requerides

    public class Classe extends Sprite
    {
        // Declaració dels atributs
                  // Aquesta variable només serà accessible des de la classe
        private var nomVariable : ClasseVariable;
                  // Aquesta variable serà accessible des de fora de la classe
        public var nomVariable : ClasseVariable;



        public function Classe()
        // Constructor de la classe (ha de dur el mateix nom que la classe)
        {

            // Inicialització dels atributs
            // Creació d'una instància de la classe
            nomVariable : ClasseVariable = new ClasseVariable();
            // Valor inicial
            nomVariable : ClasseVariable = valor;
        }
```

```
        // Declaració dels diferents mètodes de la classe
        private function metode1() {
        }
    }
}
```

Convenció de nomenclatura per a variables

És habitual l'ús de la ratlla baixa ('_') per a començar els noms de variables privades.

2.4.5. Herència i polimorfisme

1) Herència

En POO l'herència **permet crear una nova classe a partir d'una classe pare ja existent**. Es diu que aquesta nova classe hereta d'una classe pare o superclasse.

L'herència ens permet aprofitar atributs i mètodes de la superclasse i, alhora, afegir un altre tipus de funcionalitats i atributs propis a la nostra nova classe. D'aquesta manera, podem de nou aprofitar codi ja existent, objectes i propietats el comportament dels quals ens convé (i funcionen correctament) per a afegir només el que ens convingui.

En AS3 escriurem:

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

Project1 hereta de Sprite: això ens dóna accés directament a una sèrie de propietats i funcions de Sprite, com per exemple accés a propietats .buttonMode, .dropTarget, etc., o mètodes com startDrag(), stopDrag(), etc. Al seu torn, la classe Sprite hereta les propietats i els mètodes de la seva superclasse DisplayObjectContainer, i així successivament.

En general, quan fem una crida a una propietat o mètode d'una classe, es buscarà si aquesta propietat o mètode existeix en la classe mateixa; en cas contrari, es buscarà en la seva superclasse, i així successivament.

Una classe pot, al seu torn, modificar, sobreescriure un mètode de la seva superclasse (si és públic). Per a fer-ho, escriurem:

```
override public function example () {
     // Nou codi de la funció example
     }
```

que sobreescriurà la funció example() de la superclasse.

2) Polimorfisme

La idea de *polimorfisme* és que un mateix mètode (mateix nom del mètode) actuï de manera diferent en funció de la classe en què s'executa.

Imaginem que tenim dues classes Cercle i Rectangle, amb un mètode dibuixarForma que dibuixa la forma en pantalla. El polimorfisme permet que dibuixarForma() faci la seva funció malgrat que internament el codi serà molt diferent per al mètode dibuixarForma() de la classe Cercle o per al mètode dibuixarForma() de la classe Rectangle.

El polimorfisme en POO permet novament poder abstreure's del funcionament intern d'un objecte i preocupar-nos només del seu ús. En l'exemple esmentat, seria dibuixar la forma donada.

2.4.6. Associació de classes a *sprites* visuals

Flash ens permet associar fàcilment *assets* gràfics (conjunt d'elements gràfics), creats directament en Flash Professional, a una classe.

Posem per exemple que volem crear una fitxa d'un alumne. En aquesta volem mostrar informació com el seu nom, cognoms, correu electrònic, una foto de carnet, etc. D'altra banda, ens agradaria tenir un cert disseny gràfic (afegir un fons o un logotip, organitzar els elements visualment, etc.) i certa interactivitat (afegir un botó per a pujar una nova foto de perfil, per exemple).

Per començar podríem crear una classe Fitxa amb els seus atributs (nom, cognom, etc.) i mètodes corresponents (canviarFoto(), editarNom, etc.). Després, per a crear la part gràfica, podríem partir de zero, és a dir, crear tot l'aspecte gràfic per mitjà de codi, creant camps de text i situant-los en les seves coordenades respectives *x* i *y*, etc.

Això és una opció i, si el disseny és molt simple, podríem optar per aquesta via. Però Flash Professional ens ofereix un gran avantatge a l'hora de dissenyar elements. Gràcies a Flash Professional podem crear un model gràfic de la fitxa, podem situar els camps, afegir un fons, un logotip, etc. Tot això ho englobarem dins d'un símbol al qual associarem la classe que vulguem, en el nostre cas la classe Fitxa.

Aquesta idea no és nova i sovint en altres llenguatges de programació veureu englobat en un únic mapa de bits tot el material gràfic de l'aplicació. Des del codi després recuperarem els trossos de mapa de bits que necessitem i això farà que no malgastem temps redibuixant elements per mitjà del codi (sovint complicat i que sempre necessita més recursos que recuperar un simple mapa de bits).

Però Flash afegeix un avantatge suplementari. No solament podem crear l'aspecte gràfic, sinó que l'estructura del símbol creat en Flash es manté. Així, des de la classe Fitxa podrem accedir directament als seus diversos elements. N'hi haurà prou de posar el nom adequat a cada objecte en Flash Professional per tal de poder-hi accedir des de la classe associada.

Per a **associar una classe a un símbol de la biblioteca en Flash Professional** n'hi ha prou amb:

- Obrir la biblioteca de símbols (CTRL+L).
- Fer clic amb el botó dret sobre el símbol al qual volem associar la classe.
- Advanced > Export for Actionscript.
- Triar la classe a la qual volem associar aquest símbol.

Lectura recomanada

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

Nota

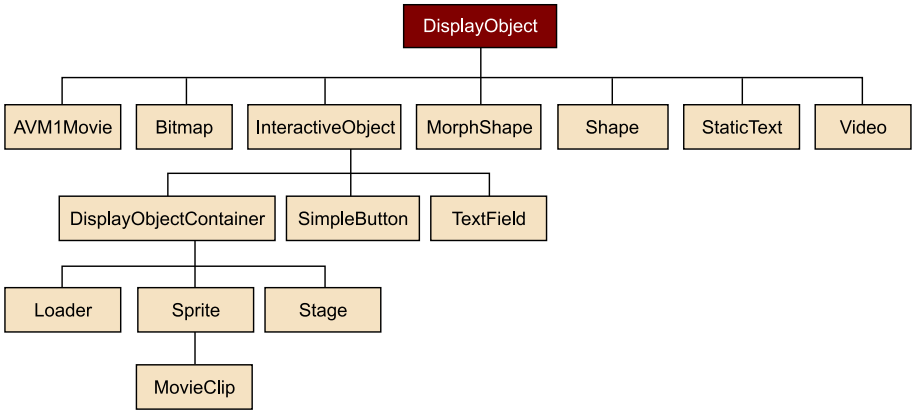
En la creació de jocs, la tècnica de tenir un mapa de bits que englobi els elements d'un joc (*blitting*) segueix sent la millor opció pel que fa a rendiment. En aquest cas, el que faríem és importar la imatge (vegeu l'apartat "Imatges") i recuperar els trossos que ens interessi per mitjà de la classe Bitmap i BitmapData.

2.5. Display List

2.5.1. Conceptes bàsics

1) La classe DisplayObject

En Actionscript tot el contingut gràfic es manipula mitjançant l'API Display. En aquesta API només un grup de classes, descendents de la classe DisplayObject, podran donar lloc a objectes representables en pantalla.



Perquè un **objecte sigui visible** hem d'haver complert **dos passos**:

- Haver creat una instància d'una de la classe DisplayObject.
- Afegir aquesta instància a la Display List (addChild()).

Només quan afegim aquest objecte a la Display List, Flash podrà mostrar-ne el contingut en pantalla.

2) La classe DisplayObjectContainer

Com podeu veure en l'arbre anterior, DisplayObjectContainer és una subclasse de DisplayObject. Aquesta classe afegeix una particularitat important: permet contenir al seu torn altres *display objects*.

Així, les instàncies de Loader, Sprite (i, per tant, MovieClip) i Stage poden, al seu torn, contenir altres instàncies de DisplayObject.

3) La Display List

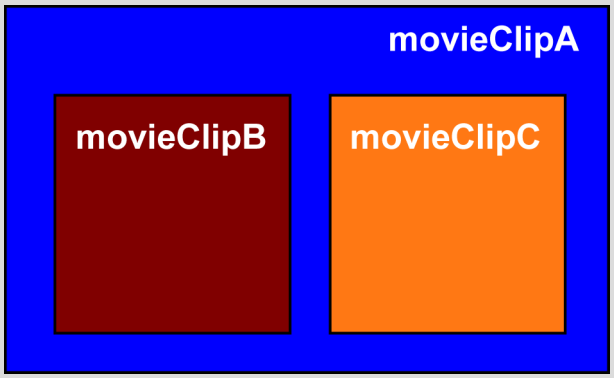
En una aplicació Flash la Display List conté tots els objectes visibles de l'aplicació. Podríem dir que és l'arbre, l'estructura visible de l'aplicació.

4) Estat inicial de la Display List d'una aplicació

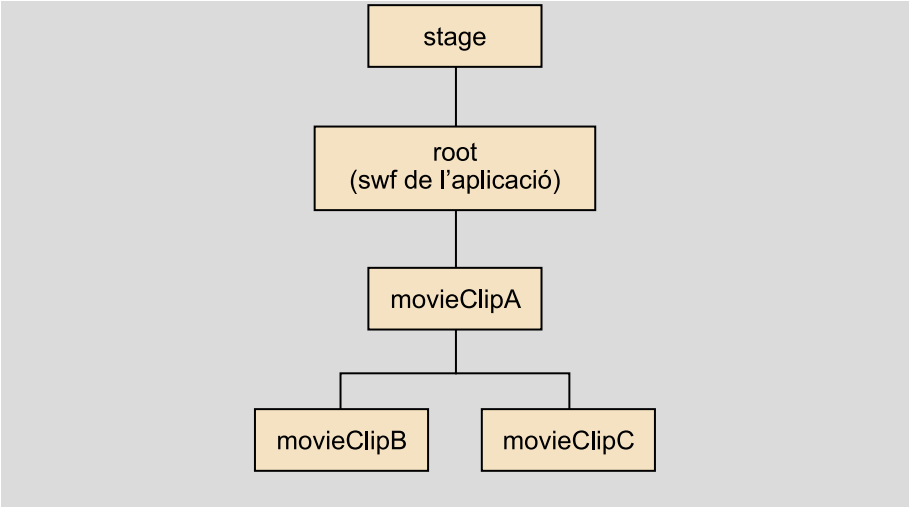
Tota aplicació Flash parteix inicialment d'una **Display List formada per**:

- Una instància **stage** de la classe Stage. Aquest objecte és la part superior de la jerarquia de la DisplayList.
- Filla d'aquest stage, tindreu **root**, una instància de la classe principal de l'arxiu swf, arrel de l'aplicació.
- En cas de tenir objectes ja creats en la línia de temps del document .fla, seran descendents directes de root.

Considerem que inicialment en una aplicació tenim un movieClipA (instància de MovieClip) i a dins tenim dos movieclips més: movieClipA i movieClipB.



Això ens donarà la DisplayList següent:



2.5.2. Treball dinàmic amb elements

Per a treballar amb la part visible de l'aplicació (visible en pantalla) haurem d'afegir (o treure) els objectes d'aquesta Display List.

1) Afegir un element a la DisplayList

Per a afegir una instància a la DisplayList farem servir **addChild**.

```
var objecte1.Sprite = new Sprite();
var objecte2:Sprite = new Sprite();
objecte1. addChild (objecte1);
```

objecte1: serà el displayObjectContainer en què volem afegir un objecte.
objecte2: serà l'objecte a afegir.

2) Modificar un element de la DisplayList

a) Opció 1: amb el nom de la instància

Si som en el mateix àmbit en què instanciem l'objecte, podem referir-nos-hi directament.

```
var objecte1:Sprite = new Sprite();
addChild (objecte1);

// desplacem l'objecte referint-nos-hi directament
objecte1.x = 200;
```

Però què passa si sortim de l'àmbit en què s'ha declarat la instància? Ja no podem referir-nos directament a l'objecte (no podrem manipular-lo o eliminar-lo). Com s'ha de solucionar aquest problema?

b) Opció 2: amb la propietat .name per a identificar un DisplayObject en la Display List de la seva Display Container.

La classe DisplayObject té la propietat .name, que identifica aquest DisplayObject dins de la Display List.

Per exemple, si dins de qualsevol àmbit fem:

```
var objecte1.Sprite = new Sprite();
objecte1.name="obj1";

// incloem l'objecte a la display list de l'aplicació:
root.addChild(objecte1);
```

Ara podrem recuperar el control de l'objecte directament per mitjà de la Display List. D'una banda, tenim accés a la Display List de root i de l'altra farem servir la propietat .name per a localitzar l'objecte que ens interessa. Llavors utilitzarem el mètode

`DisplayObjectContainer.getChildByName(nameChild)`

Per exemple, si escrivim:

```
// Gràcies a la propietat .name podem localitzar l'objecte
// en la display list
root.getChildByName("obj1").x = 100;
```

3) Ús i funcionament d'index en la DisplayList

Quan afegim objectes a la Display List (a un mateix Display Object Container), quedaran en el mateix nivell de la Display List (vegeu l'arbre), però visualment s'aniran superposant un sobre l'altre. L'objecte que aparegui al dessota de tots tindrà index = 0, mentre que a mesura que anem cap endavant, tindrem index = 1, 2, 3, etc.

Tal com fèiem amb la propietat .getChildByName, podrem recuperar l'objecte que es trobi en un índex concret gràcies al mètode

`DisplayObjectContainer.getChildAt(numIndex)`

```
var objecte1.Sprite = new Sprite();
var objecte2.Sprite = new Sprite();
var objecte3.Sprite = new Sprite();

// incloem els objectes a la display list de l'aplicació:
addChild(objecte1);  // ---> index = 0
addChild(objecte2);  // ---> index = 1
addChild(objecte3);  // ---> index = 2

// si volem moure el fill en l'índex 1, objecte2, farem
getChildAt(1).x = 100;  // mourà l'objecte2 a la posició x = 100
```

Informacions pràctiques:

`getChildAt(0)`: serà sempre l'objecte situat més enrere.

`getChildAt (numChildren()-1)`: serà l'objecte situat més endavant.

`numChildren` indica el nombre de fills d'una displayContainer.

4) Treure un element de la DisplayList

Per a treure una instància de la DisplayList podem:

- Coneixent el nom de la instància, fer nomInstancia. **removeChild()** (o bé removeChild(nomInstancia)).

- Coneixent-ne el nom fer **getChildByName(childName).removeChild()** (o bé removeChild(getChildByName(childName))).

- Coneixent-ne l'índex farem servir **removeChildAt**(index).

Atenció, quan eliminem un objecte de la Display List, els objectes supliran (ompliran) aquest espai buit. Mai no queda un índex buit, sempre tindrem els índexs ordenats successivament 0,1,2... marcant els diferents nivells de profunditat.

```
var objecte1.Sprite = new Sprite();
var objecte2.Sprite = new Sprite();
var objecte3.Sprite = new Sprite();
var objecte4.Sprite = new Sprite();

// incloem els objectes a la display list de l'aplicació:
addChild(objecte1);  // ---> index = 0
addChild(objecte2);  // ---> index = 1
addChild(objecte3);  // ---> index = 2
addChild(objecte4);  // ---> index = 3

// si esborrem l'objecte2
  removeChild(objecte2);

  // els índexs es reorganitzen per a ocupar aquest buit
// objecte1 ---> index = 0
// objecte3 ---> index = 1
// objecte4 ---> index = 2
```

Un altre punt molt important que cal tenir en compte és que en cap cas estarem esborrant l'objecte, simplement el traiem de la Display List.

Si per exemple el traiem de la Display List de l'aplicació (Display List visible), l'objecte deixarà de veure's en pantalla, però no deixarà d'existir.

5) Intercanviar la profunditat entre dos fills

Utilitzant directament cada objecte podem fer:

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

O podem fer-ho utilitzant els índexs de cada objecte:

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

6) Desplaçar-nos dins de la Display List

- `.parent`: per a referir-nos a l'objecte immediatament superior en la jerarquia.
- `.nomDelFill`: per a referir-nos al descendent amb el nom "nomDelFill".
- `this`: per a referir-nos a l'objecte actual.

És important no confondre la jerarquia dins de la DisplayList amb la jerarquia existent entre classes (classes descendents i superclasses).

Bibliografia

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

2.6. Model d'esdeveniments

2.6.1. Conceptes bàsics

Segons la Viquipèdia:

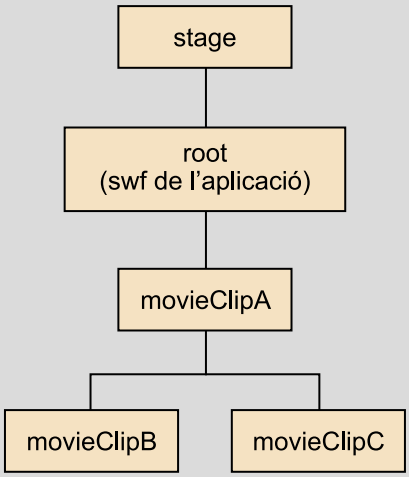
> "La programació dirigida per esdeveniments és un paradigma de programació en el qual tant l'estructura com l'execució dels programes són determinats pels successos que tenen lloc en el sistema."

A diferència de la programació seqüencial, en la qual es van executant instruccions pas a pas, en la programació dirigida per esdeveniments s'executaran una sèrie d'instruccions només quan tingui lloc un esdeveniment.

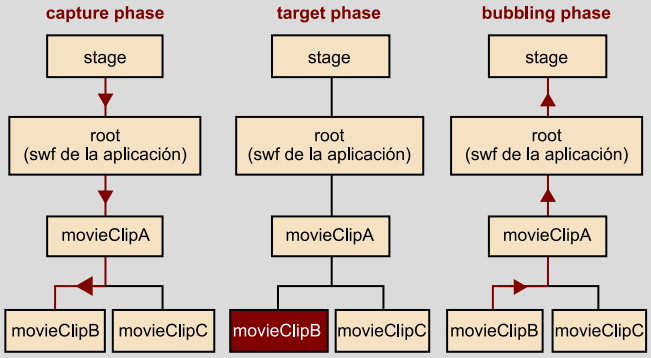
> Un exemple d'esdeveniment podria ser haver fet clic amb el ratolí o qualsevol altra interacció de part de l'usuari, però també pot ser que s'hagi acabat la càrrega de dades des d'un servidor o la càrrega d'una imatge.

En AS3 cada objecte té els seus esdeveniments propis. Flash Player (o Air) s'ocupa de gestionar-los automàticament. D'aquesta manera, en resposta a un succés crearà un objecte, instància de la classe Event (o d'una classe hereva d'Event), que distribueix l'objecte font d'aquest succés. Si aquest objecte font forma part de la DisplayList, l'objecte esdeveniment es distribuirà seguint la jerarquia de la DisplayList. En alguns casos (*bubbling phase*), aquest objecte esdeveniment es transmetrà de tornada per la jerarquia de la DisplayList. Aquest recorregut per la DisplayList s'anomena **EventFlow**.

Recordem l'exemple vist en el tema de la DisplayList. Aquesta aplicació simple conté en la línia de temps un movieclipA i, al seu torn, dins d'aquest, dos movieclips B i C. La DisplayList seria la següent.



Quan fem clic sobre el movieclipB, Flash (o Air) distribuiran una instància de MouseEvent que seguirà el recorregut següent:



Nota

Veiem que tot i que el clic es realitzi sobre el movieClipB, la instància de MouseEvent recorrerà els diferents elements de la DisplayList (des de *stage* fins a l'element clicat). Aquest recorregut de l'esdeveniment ens aporta un gran avantatge, ja que permet centralitzar el codi de gestió d'esdeveniments en un únic objecte. Per exemple, en l'estructura anterior, podem veure que des de root podem detectar un clic sobre movieClipA, B o C, atès que l'esdeveniment MouseEvent.CLICK es distribuirà sempre passant per root (en la fase de captura o en la fase *bubbling*).

Enllaços relacionats

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

"Clase Event" (documentació Adobe).

2.6.2. Gestió dels esdeveniments en AS3

Fins ara hem vist com Flash actua i gestiona automàticament la distribució d'esdeveniments, i ens dóna informació en tot moment del que passa en el sistema.

Ara, com a programadors, ens interessa poder detectar aquests esdeveniments per a executar les instruccions que ens interessi. El que volem és escoltar un esdeveniment concret per a executar una funció particular.

Per exemple, podríem escoltar els esdeveniments emesos quan es pressionen les tecles de direcció per tal de moure el nostre personatge en un joc.

Per a fer-ho, haurem de tenir en compte **tres dades essencials**:

- **Quin objecte estarà escoltant l'esdeveniment**.
- **Quin esdeveniment volem detectar o escoltar**.
- **Quina acció volem que s'executi si es produeix l'esdeveniment sobre aquest objecte**.

Advertiment

Com hem vist, en la detecció del clic sobre un element, no té perquè ser aquest mateix element el que escolta. Un element pare podrà ser més útil per a centralitzar la detecció dels clics sobres els seus elements fill, per exemple.

AS3:

```
objecteQueEscolta.addEventListener(esdevenimentAEscoltar,funcioAExecutar);
```

Nota

Per defecte, addEventListener escoltarà solament la fase *bubbling* del recorregut de l'esdeveniment. Per exemple, per a un esdeveniment MouseEvent.CLICK escoltarem el recorregut des de l'element clicat fins a l'arrel, stage, de l'aplicació. Però podem escoltar la fase de captura posant:

```
objecteEnQueRecauLEsdeveniment.addEventListener(MouseEvent.CLICK,funcioAExecutar,
true);
```

2.6.3. Esdeveniments comuns

Per norma general, cada classe tindrà els seus esdeveniments propis, de manera que és molt aconsellable fer sempre una ullada a la documentació d'Adobe per a veure quins esdeveniments tenim a la nostra disposició. La llista de possibles esdeveniments és llarga i variada; vegem-ne alguns dels més habituals.

Interacció amb l'usuari

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

Són els referents a la interacció amb el teclat:

- Tecla pressionada (KEY_DOWN)
- Tecla deixada de pressionar (KEY_UP)

Per a poder escoltar aquest esdeveniment, l'element que escolti ha de tenir el focus; per això s'aconsella que sigui stage qui escolti aquests esdeveniments:

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

Nota

Si escoltéssim aquest esdeveniment des d'un objecte de l'aplicació (i no des de stage), per a detectar la interacció amb el teclat, abans hauríem de donar-li el focus a l'objecte, per exemple fent-hi clic.

2) Esdeveniments de ratolí (MouseEvent)

Esdeveniments relacionats amb la interacció amb el ratolí. Cal destacar que una acció del ratolí pot provocar més d'un esdeveniment. Per exemple, el clic del ratolí Flash enviarà un esdeveniment MOUSE_DOWN i també MOUSE_UP i MOUSE_CLICK. Això ens permet més possibilitats i un control més gran de les interaccions.

Entre altres tindrem:

- El clic esquerre amb el ratolí (CLICK) però també el central (MIDDLE_CLICK) o dret (RIGHT_CLICK).
- Doble clic del ratolí (DOUBLE_CLICK).
- Detecció de l'estat del ratolí, pressió del ratolí (MOUSE_DOWN), deixar de pressionar (MOUSE_UP). Molt útil quan es fan aplicacions en què podem fer clic i mantenir el botó polsat per a desplaçar objectes (arrossegar i deixar anar, *drag and drop*).
- Moviment del ratolí (MOUSE_MOVE).

3) Esdeveniments per a dispositius tàctils (TouchEvent)

Orientats a aplicacions en Air i per a pantalles tàctils (telèfons intel·ligents, tauletes), tenim detecció d'esdeveniments tàctils com ara:

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

Enllaç relacionat

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

O també detecció de gestos:

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

Nota

Per a poder fer servir la detecció de gestos no s'ha d'oblidar incloure abans:

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

Altres esdeveniments d'interès

1) Event.ENTER_FRAME

L'esdeveniment Event.ENTER_FRAME es pot escoltar des de qualsevol instància de la classe DisplayObject i s'emet automàticament cada 1/velocitatDeLAnimacióFlash. És a dir, si la nostra aplicació corre a 30 marcs/segon, podrem escoltar l'Event.ENTER_FRAME cada 1/30 segons.

Aquesta característica el fa molt útil per a les transicions, bucles en què incrementem una propietat en cada iteració (cada 1/frameRate).

Observació

De fet, molts paquets d'interpolacions, *tweens* –que veurem en l'apartat "Animació per a programació: *tweens*"– funcionen internament utilitzant aquest esdeveniment.

2) TimerEvent

Per a utilitzar TimerEvent crearem una instància de la classe Timer. El funcionament és similar a Event.ENTER_FRAME, però amb un grau de control més alt, atès que podem triar el temps entre esdeveniments i el nombre de vegades que es repetirà.

És molt pràctic si volem realitzar una acció a intervals de temps precisos.

Podríem, per exemple, crear un rellotge virtual i gràcies a Timer moure la busca dels segons a cada segon (optimitzant millor els recursos que si féssim servir Event.ENTER_FRAME).

Monitoratge de la transferència de dades

1) Event.INIT i Event.COMPLETE

S'utilitza, per exemple, per a controlar la inicialització de la càrrega de dades i el fet que la càrrega hagi finalitzat.

2) ProgressEvent

Per a monitorar el desenvolupament de la descàrrega.

Vegeu també

Veurem amb més detall aquest aspecte en l'apartat "*Loaders*".

2.6.4. Esdeveniments personalitzats

En la majoria de casos, els esdeveniments existents són més que suficients per a qualsevol aplicació, però hi ha moments en què potser vulguem crear el nostre propi esdeveniment personalitzat.

És més, un esdeveniment personalitzat ens permet, si creem una altra classe hereva d'Event, afegir atributs personalitzats, propietats que l'esdeveniment transportarà i que no tenen els esdeveniments existents.

En general, per a l'ús d'esdeveniments personalitzats haurem de:

- Crear una nova classe d'esdeveniment descendent d'Event, amb els seus nous atributs o mètodes, si volem que tingui noves propietats o funcions.

- Tenir un objecte descendent de la classe EventDispatcher, per a poder emetre esdeveniments.

Observació

Cal saber que la classe DisplayObject és descendent d'EventDispatcher, de manera que qualsevol displayObject podrà, per defecte, emetre un esdeveniment.

AS3:

Podem emetre un esdeveniment personalitzat sense crear una nova subclasse d'Event fent:

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

2.6.5. Aplicacions adaptades a múltiples pantalles

En l'actualitat, una de les dificultats més grans a l'hora de planificar el disseny d'una aplicació és la multitud de dispositius en què aquesta es podrà visualitzar. És més, ja no ens enfrontem només diferents dispositius, sinó que un mateix dispositiu pot tenir dos modes de visualització.

> Un telèfon intel·ligent o una tauleta, per exemple, es podrà veure en vertical o apaïsat, la qual cosa implica una redisposició dels elements gràfics.

La nostra aplicació ha de poder adaptar-se a aquests canvis i per això podem utilitzar dos esdeveniments en particular:

1) Event.RESIZE

L'esdeveniment Event.RESIZE és especialment interessant en aplicacions d'escriptori, i s'emet cada vegada que es modifiquen les dimensions de la finestra del Flash Player. Per a poder escoltar aquest esdeveniment caldrà ferho des de stage:

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

Qualsevol canvi de la grandària executarà la funció resizeHandler(). Aquesta funció, per exemple, podria tenir en compte les noves dimensions de la finestra per a resituar els botons de manera diferent (una mica com faríem en el web amb tècniques de *responsive design*).

Algunes **propietats útils per a treballar amb la grandària del stage** són:

- **`stage.scaleMode`**: mode en què actua Flash quan es canvia la grandària de la finestra; per exemple, si posem

```
stage.scaleMode = StageScaleMode.NO_SCALE;
```

els elements dins de Flash no canviaran de grandària en canviar la grandària de la finestra.

- **`stage.align`**: alineació de l'stage en relació amb la finestra del lector Flash; habitualment el més pràctic és posar
  `stage.align = StageAlign.TOP_LEFT;`

- **`stage.stageWidth`**: ens donarà l'amplada de la nostra finestra.

- **`stage.stageHeight`**: ens donarà l'altura de la nostra finestra.

2) StageOrientationEvent

És especialment adequat per a dispositius mòbils i AIR2 (i versions superiors): farà que l'objecte Stage emeti dos tipus d'esdeveniment quan hi hagi un canvi d'orientació del dispositiu:

- **`StageOrientationEvent.ORIENTATION_CHANGING`**: indicarà que l'orientació del dispositiu està canviant.

- **`StageOrientationEvent.ORIENTATION_CHANGE`**: indicarà que l'orientació del dispositiu ha canviat; les propietats `.beforeOrientation` i `.afterOrientation` ens informaran de l'orientació anterior i de l'actual.

Lectures recomanades

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

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

2.7. Treball amb dades externes

Fins ara hem tractat aplicacions Rich Media amb tot el contingut intern. El funcionament de l'aplicació es realitzava amb un contingut donat, inclòs en les classes, fonts .swf o fruit de certa interacció amb l'usuari, però sense intercanvi de dades amb l'exterior. En cas de voler canviar el contingut, canviaríem el codi, creant una nova aplicació.

Però aquest mètode, que podria ser adequat per a aplicacions simples que no hagin de canviar mai, no és l'habitual.

En la realitat actual, una aplicació demanda un contingut dinàmic, un contingut actualitzat i aplicacions flexibles, amb un cert marge de parametrització, però sense haver de crear cada vegada una aplicació nova.

Sovint connectarem amb una base de dades que ens subministrarà informació actualitzada o podrem parametritzar externament determinades propietats de la nostra aplicació, sense haver de compilar de nou l'aplicació.

2.7.1. Mètodes de càrrega de dades

Per a començar, haurem de recuperar les dades, carregar les dades situades en un servidor remot o localment.

En AS3 usarem principalment per a aquest objectiu dues classes que ens facilitaran aquesta càrrega de dades:

- La classe **UrlLoader**: especialment orientada a la càrrega de dades de text o binàries, com poden ser fitxers XML o JSON, imatges, swf, etc.

- La classe **Loader**: especialment orientada a la càrrega de .swf i fitxers gràfics externs.

I seguirem els passos següents:

- Càrrega **de les dades**.

- **Interpretació de les dades**, *bytes*, en un objecte que puguem manipular des del codi.

2.7.2. Formats d'intercanvi de dades

A l'hora d'intercanviar dades, les primeres preguntes serien: quin format utilitzar?, com estructurar la informació?, com trobar un format comú d'ús per a qualsevol intercanvi de dades?

> La **serialització de la informació** és el procés de codificar un objecte per a poder ser desat (en arxiu o en memòria intermèdia) i enviat electrònicament, cosa que permet l'intercanvi de dades.

Però el fet de poder tractar les dades no seria suficient sense un format mínimament llegible, un format que ens permeti una bona abstracció de l'estructura de dades; és a dir un format que ens permeti entendre l'estructura de les dades d'un cop d'ull.

XML i, més recentment, **JSON** són dos formats que ens permeten aquesta serialització i que, a més, permeten obtenir un format llegible.

Habitualment tindrem una base de dades en un servidor, amb tota la informació necessària. Gràcies a un llenguatge del costat del servidor, com podria ser PHP, fent una petició a la base de dades obtindrem només la informació necessària. Aquesta informació final, generalment en format XML o JSON, és la que rebrà la nostra aplicació i amb la qual treballarà.

2.7.3. Treball amb XML i JSON

Treball amb XML

L'ús de XML per a la serialització de dades es remunta a molt temps enrere, de manera que Flash incorpora ja una sèrie de classes que ens permeten la lectura i la manipulació de fitxers XML de manera simplificada.

Enllaç relacionat

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

1) Estructura d'un XML

La tecnologia XML estructurarà la informació gràcies a elements niats. Desarem les dades d'una de les formes següents:

- Com a contingut d'un node.
- Com a atribut d'un node.

I estructurarem la informació niant els nodes adequadament.

Suposem que volem catalogar una biblioteca amb una estructura XML. De cada llibre tenim un codi ISBN, el títol del llibre i el nom de l'autor. Una estructura XML per a aquest cas podria ser:

```
<?xml version="1.0" encoding="UTF-8"?>
<BIBLIOTECA>
    <LLIBRE isbn = "codi1">
        <TITOL>títol 1</TITOL>
        <AUTOR>autor 1</AUTOR>
    </LLIBRE>

    <LLIBRE isbn = "codi2">
        <TITOL>títol 2</TITOL>
        <AUTOR>autor 2</AUTOR>
    </LLIBRE>

    <LLIBRE isbn="codi3">
        <TITOL>títol 3</TITOL>
        <AUTOR>autor 3</AUTOR>
    </LLIBRE>
</BIBLIOTECA>
```

Veiem que per a cada llibre desem el codi ISBN com a atribut d'un node llibre. D'altra banda, cada node llibre té dos nodes fill <titol> i <autor>, que contenen la informació del títol del llibre i el seu autor, respectivament.

L'estructura XML següent podria també estructurar la mateixa informació:

```
<?xml version="1.0" encoding="UTF-8"?>
<BIBLIOTECA>
<LLIBRE isbn="codi1" titol="títol 1" autor="autor 1"></LLIBRE>
<LLIBRE isbn="codi2" titol="títol 2" autor="autor 2"></LLIBRE>
<LLIBRE isbn="codi3" titol="títol 3" autor="autor 3"></LLIBRE>
</BIBLIOTECA>
```

2) Lectura de dades XML

a) Recuperació del fitxer XML

El primer que haurem de fer és recuperar, carregar, el fitxer extern. Per a això farem servir la classe **URLLoader** nativa d'AS3 i **URLRequest** per a indicar la URL on es trobi l'arxiu.

Inicialitzem les variables:

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

Carreguem les dades:

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

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

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

Nota

No cal que el fitxer .xml existeixi físicament. De fet, en la majoria dels casos farem una crida a un servei web que ens donarà directament com a resposta la informació en format XML. Així, serà més habitual tenir alguna cosa com:

```
url:URLRequest = new URLRequest ("webService.php?parametre1=valor1")
```

on webService és una funció php que ens dóna, a partir de certs paràmetres de recerca, una resposta en format XML.

b) Recuperació de la informació (anàlisi sintàctica del fitxer XML)

Una vegada hem fet la càrrega del fitxer XML, n'obtenim el contingut. Però aquest contingut ha estat llegit i recuperat com una sèrie de caràcters, com un String, no com una estructura de dades. Ens falta interpretar aquest String per a poder-lo organitzar en un objecte i poder-lo manipular amb facilitat posteriorment.

Per a això AS3 té l'avantatge d'estar proveït de classes natives per a XML. Sabent que el que estem tractant és l'estructura d'un document XML, aquestes classes ens permeten interpretar aquest String i transformar-lo en un objecte estructurat.

N'hi haurà prou de crear un objecte XML a partir de l'String recuperat:

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

c) Navegar dins de l'estructura del nou objecte XML

Explorar un objecte XML és molt semblant a explorar una estructura d'arxius amb les seves conseqüents carpetes.

Partint de l'estructura XML de la biblioteca (versió1), vegem algunes instruccions d'interès que podríem utilitzar:

Obtenir els fills (llista de llibres que componen la biblioteca):

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

o, atès que ja coneixem l'estructura, podríem escriure:

```
llibres:XMLList = xml.LLIBRE;
```

Lectura del valor d'un atribut d'un node (per exemple, autors dels llibres):

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

o, el que és el mateix,

```
autors:XMLList = xml.LLIBRE.AUTOR;
```

Lectura del contingut d'un node:

Utilitzarem un índex per a escollir el fill al qual volem referir-nos.

Així, si volem accedir a l'autor del primer llibre de la biblioteca escriurem:

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

Nota

0 es referirà al primer dels fills.

Lectura d'un atribut d'un node:

Per a això utilitzarem @ per a indicar l'atribut que volem llegir. Seguint amb l'exemple de la biblioteca, si volem llegir el codi ISBN del primer llibre, farem:

```
isbnPrimerLlibre:String = xml.LLIBRE[0].@isbn;
```

Exploració de l'objecte XML:

Sovint no sabrem quants elements componen el nostre XML. Haurem d'explorar l'estructura passant de fill a fill i extreure'n la informació. Amb aquest objectiu, hi ha diverses funcions que ens seran d'utilitat:

- **Funció length()**. Ens donarà el nombre de nodes que compon un objecte XMLList. Així, en l'exemple, per a obtenir el nombre de llibres escriurem:

```
numLlibres:int = xml.LLIBRE.length()
```

- **Funció parent()**. Per a accedir al node pare.

- **Funció nextSibling** i **funció previousSibling**. Per a passar al germà adjacent, posterior o anterior, respectivament.

- **ChildIndex()**. Per a conèixer l'índex d'un node.

Lectura recomanada

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

Treball amb JSON

A causa sobretot d'una representació més bona de la informació i de la seva facilitat de processament i codificació, JSON s'ha convertit en poc temps en un dels formats més utilitzats en transmissió de dades, reemplaçant sovint l'XML.

Enllaços relacionats

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

Validador JSON. http://jsonlint.com/

1) Estructura en JSON

Bàsicament tenim dues estructures possibles:

- Sèries de parells nom / valor. Similar a l'estructura d'un Associative Array en AS3.

- Llistes de valors. Similar a Array en AS3.

Com veiem, JSON utilitza convencions d'escriptura molt semblants a les de la majoria de llenguatges de programació moderns.

En l'exemple de la biblioteca, la versió JSON seria la següent:

```
{"llibre": [

   {
   "isbn": "codi1",
   "titol": "títol1",
   "autor": "autor1"
   },

   {
   "isbn": "codi2",
   "titol": "títol2",
   "autor": "autor2"
   },

   {
   "isbn": "codi3",
   "titol": "títol3",
   "autor": "autor3"
   }

   ]
}
```

Nota

Fa ben poc s'han afegit classes natives directament en Flash per a la manipulació JSON: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/JSON.html.

Però també hi ha biblioteques de tercers, com ara https://github.com/mikechambers/as3corelib, que han estat fins ara una alternativa de qualitat per a treballar amb JSON.

2) Passos a seguir per a la lectura d'un fitxer JSON utilitzant as3corelib

a) Lectura del fitxer

Igual que per a la lectura de dades XML, utilitzarem URLLoader (i el seu **URLRequest**) per a recuperar les dades.

Inicialització de les variables:

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

Càrrega de les dades:

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

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

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

b) Interpretació de l'String obtingut i creació d'un objecte estructurat amb què treballar (anàlisi estructural del JSON)

Una vegada carregada la informació, podrem recuperar-la en dataLoader.data. Però atenció, aquesta informació se'ns presenta de nou com una cadena de caràcters, un String. Ara ens cal interpretar aquesta informació i estructurar-la en un objecte que ens pugui ser d'utilitat.

Per a això, utilitzarem la biblioteca as3corelib, que podem trobar a: https://github.com/mikechambers/as3corelib/downloads i, més concretament, farem servir el paquet com.adobe.serialization.json.

Després d'afegir el paquet a la nostra aplicació, haurem d'importar la classe que ens interessa:

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

A continuació utilitzarem la funció JSON.decode() per a obtenir l'objecte estructurat a partir de l'String carregat:

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

Nota

En la nova classe nativa JSON d'AS3, aquesta acció es realitza amb la funció JSON.parse() (Atenció: no s'ha de confondre JSON nativa amb JSON as3corelib).

c) Navegació i extracció de la informació a partir de l'objecte JSON

L'extracció d'informació seria molt semblant a la que faríem en un sistema d'Arrays. D'una banda, utilitzarem un índex entre [ ] per a seleccionar un element dins d'un vector. Després, amb .nomAtribut obtenim el valor de l'atribut en qüestió.

Seguint amb l'exemple de la biblioteca (versió JSON), podríem:

Obtenir tots els llibres amb:

```
data.llibre
```

Obtenir el primer llibre amb:

```
data.llibre[0]
```

Obtenir l'autor del primer llibre amb:

```
data.llibre[0].autor
```

Obtenir el nombre de llibres de la biblioteca (equivalent a obtenir el nombre d'elements de l'Array llibre) **amb**:

```
data.llibre.length;
```

Atenció: no s'ha confondre la propietat .length amb la funció length() utilitzada en l'objecte XMLList.

2.7.4. *Preloaders*

Acabem de veure que el tractament de dades, en XML, JSON o altres formats, seguirà sempre uns passos semblants:

- La petició al servidor de les dades que necessitem.
- Espera de resposta del servidor. El servidor processa la nostra petició.
- S'estableix la connexió amb el servidor i es recuperen les dades.
- S'interpreten les dades rebudes.

Veiem que tant per al segon com per al tercer pas, l'aplicació estarà en espera, és a dir, estarem esperant una resposta per tal de, després, descarregar les dades. Aquests temps d'espera dependran tant de la connexió com de la resposta del servidor o de la quantitat d'informació que estiguem descarregant, i poden arribar a ser considerables.

D'altra banda, l'usuari, sense coneixement del que està passant, esperarà una resposta immediata a la seva acció; veient que no ocorre immediatament pot pensar:

- Que no s'ha detectat el clic (i, per tant, intentarà fer clic de nou i llençarà novament la petició).

- Pitjor encara, que l'aplicació no funciona i l'abandonarà.

És per això que sempre que tinguem una càrrega de dades o espera, caldrà indicar-ho. I aquí rau la utilitat principal dels ***preloaders*** (precarregadors), que és indicar a l'usuari que està succeint alguna cosa.

Podem diferenciar **dos tipus de *preloaders***:

a) Un ***preloader* no quantitatiu** (més habitual en la fase 2, temps d'espera d'una petició). Si no podem estimar el temps d'espera (com passa per a la resposta del servidor), utilitzarem una petita animació que es repeteixi (per a indicar que l'aplicació no està bloquejada) i un text explicatiu que indiqui què està passant.

Per posar un exemple, una aplicació en què fem una crida a la base de dades per a recuperar informació, podria mostrar el típic missatge "Recuperant informació de la base de dades..." amb el típic *spinner* donant voltes.

b) Un ***preloader*** **quantitatiu** (més habitual en la fase 3, fase de descàrrega). Sempre que tinguem coneixement del pes final podem indicar amb una barra de progressió (o un altre element gràfic) el percentatge de dades ja descarregat. Aquest tipus de *preloader* té l'avantatge d'informar a l'usuari sobre el temps estimat que falta.

> Un exemple molt habitual el podem trobar en descàrregues de vídeo, on una barra de progressió ens indica que la descàrrega està funcionant i fins on s'ha descarregat en aquest moment.

Un dels avantatges en AS3 és que les classes de càrrega de dades, URLLoader o Loader, ens permeten escoltar esdeveniments dedicats i conèixer les diferents etapes de la descàrrega. D'aquesta manera, no solament podem saber el final d'una descàrrega (com ja hem vist en la càrrega de dades XML o JSON amb URLLoader), sinó que podem saber quan comença, seguir-ne la progressió, si hi ha errors, etc.

Vegem alguns dels esdeveniments bàsics per a l'ús de *preloaders* i la lleugera diferència de funcionament entre URLLoader i Loader.

1) URLLoader

En aquest cas, el funcionament és força directe, ja que podem escoltar directament els esdeveniments des d'URLLoader:

- **Event.open**: indica l'inici de l'operació després de llançar la descàrrega URLLoader.load().

- **Event.progress**: es rep quan estem rebent dades de la descàrrega; durant aquesta etapa podrem recuperar informació sobre la descàrrega gràcies a les propietats de l'objecte URLLoader:
  - **.bytesLoaded**: indica el nombre de *bytes* descarregats.
  - **.bytesTotal**: indica el nombre total de *bytes* de la descàrrega final.

- **Event.complete**: Indica que s'han descarregat totes les dades.

Com hem vist en el capítol anterior, és en aquest punt en el qual podem processar les dades rebudes.

2) Loader

Malgrat que el funcionament és molt semblant al d'URLLoader, hi ha una diferència que cal tenir en compte.

En el cas de Loader, igual que amb URLLoader, iniciarem la descàrrega amb Loader.load(), però l'escolta d'esdeveniments i la recuperació de les dades no es farà directament a través de Loader, sinó de la seva propietat Loader.contentLoaderInfo.

Així, serà des d'URLLoader.contentLoaderInfo que escoltem:

- **Event.open**: inici de la descàrrega.

- **Event.progress**: progressió de la descàrrega; en aquesta fase, com hem vist anteriorment, usarem:
  - **.bytesLoaded**: indica el nombre de *bytes* descarregats.
  - **.bytesTotal**: indica el nombre total de *bytes* de la descàrrega final.

- **Event.complete**: indica que s'ha descarregat tota la informació.

- **Event.init**: quan descarreguem un .swf, ens permetrà tenir accés a propietats del .swf final encara que no s'hagi acabat la descàrrega.

Per exemple, podríem accedir a propietats com l'amplada i l'altura abans que s'acabi la descàrrega.

Nota

L'esdeveniment Init sempre es produeix abans que l'esdeveniment Complete.

Hem vist els diferents tipus de *loaders* i els seus esdeveniments, vegem-ne ara l'aplicació amb exemples pràctics.

2.7.5. Imatges

En la descàrrega d'imatges utilitzarem la classe Loader.

Vegem-ne un exemple pas a pas:

1) Iniciarem la càrrega (afegint els escoltadors d'esdeveniments que ens interessi):

```
//instanciació de Loader i URLRequest
var loader:Loader = new Loader();
var urlImage:URLRequest = new URLRequest("lanostraImatge.jpg");

//Creació d'escoltadors
//Atenció, observeu que els escoltadors s'apliquen sobre
//loader.contentLoaderInfo
loader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS,
                                          progressHandler);
loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
                                          completeHandler);

//Comencem la descàrrega
loader.load(urlImage);
```

2) Tractarem els esdeveniments:

Amb **ProgressEvent.PROGRESS** podrem controlar la descàrrega, veure quant s'ha descarregat:

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

  // Podem veure els bytes carregats
  trace("Bytes carregats : " + e.bytesLoaded);

  // Veure els bytes totals
  trace("Bytes carregats : " + e.bytesTotal);

  // O fer el càlcul del percentatge carregat
  trace("Percentatge carregat : " + (e.bytesLoaded/e.bytesTotal)*100
  + "%");

}
```

És aquí on podríem fer que un objecte *preloader* indiqués, per mitjà d'una barra de progressió, el percentatge carregat.

Finalment, amb **Event.COMPLETE** detectarem que la descàrrega ha finalitzat i podrem processar la imatge.

La classe Loader és una classe descendent de DisplayObjectContainer amb una particularitat important: només pot contenir un display object en la seva Display List. Així, si utilitzem la mateixa instància de Loader per a carregar una imatge i després una segona imatge, la segona imatge suprimirà la primera.

Una manera de solucionar això (si no volem crear en cada càrrega d'imatge una nova instància de Loader) serà crear un objecte intermediari en el qual bolcarem la imatge tot just carregada.

Lectura recomanada

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

Vegem-ho pas a pas:

```
private function completeHandler(e:Event):void {


// La càrrega de la imatge ha finalitzat
trace("IMAGE LOADING COMPLETED");


// Ara tenim accés a les propietats de la imatge
// Podríem, per exemple, recuperar informació sobre les dimensions de la imatge
```

```
var imageW = e.target.width;
var stageW = stage.stageWidth;



// Suposem que hem creat una variable global imageContainer instància de Sprite
// Podem anar posant les imatges carregades dins de la displayList d'imageContainer
imageContainer.addChild(e.target.content);
// Recordeu que en fer addChild sobre un altre objecte, el display object amb la imatge
// desapareixerà de la display list de Loader
// De manera que podem tornar a carregar una altra imatge amb la mateixa instància
// de Loader.

// Recordeu que si volem visualitzar la imatge en pantalla caldrà afegir imageContainer
// a la display list de l'aplicació addChild(imageContainer);
}
```

2.7.6. Àudio

En la importació i manipulació d'àudio utilitzarem una **classe principal Sound**, que s'ocuparà de la descàrrega del so. És la classe Sound la que s'ocuparà tant de la descàrrega (*load*) i de l'inici de la lectura (*play*), com del fet de tancar l'*stream* (*close*), és a dir, anul·lar la descàrrega.

La manipulació del so es realitzarà mitjançant la classe SoundChannel. Per a això, quan utilitzem el mètode play() es genera un nou objecte SoundChannel alhora que es llança la lectura d'àudio. A partir d'aquesta nova instància, aquest objecte SoundChannel, podrem (entre altres coses):

- Obtenir informació sobre el volum actual i la posició de lectura.
- Aturar la lectura gràcies al mètode stop().
- Manipular el so per mitjà d'un objecte SoundTransform.

Resumint, el tractament de so es reparteix entre:

- **La classe Sound**. Permet treballar amb so en una aplicació. En concret, la classe Sound permet crear un objecte Sound, carregar i reproduir un arxiu MP3 extern en l'objecte, tancar el flux de so i accedir a dades de so com, per exemple, informació sobre el nombre de *bytes* del flux i les metadades ID3.

- **La classe SoundChannel**. SoundChannel ens ajudarà a controlar el so en una aplicació. Cada so està assignat a un canal de so i l'aplicació pot tenir diversos canals de so que s'estiguin llegint alhora. La classe SoundChannel conté un mètode stop(), propietats per a supervisar l'amplitud (volum) del canal i una propietat per a assignar un objecte SoundTransform al canal.

Vegem pas a pas **com funcionaria la importació i la lectura d'un fitxer àudio extern**.

1) Descàrrega i monitoratge (classe Sound)

Per a començar necessitarem instàncies de Sound, URLRequest i 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();
```

Iniciarem la càrrega del so, però abans, igual que amb qualsevol altra càrrega externa, afegirem diferents escoltadors d'esdeveniments que ens poden ser útils:

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


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

El mètode load de la classe Sound té un funcionament molt similar al load de Loader. Ens permet, mitjançant l'ús d'esdeveniments, monitorar la descàrrega de so. Podríem, per exemple, seguir el progrés de la descàrrega o comprovar si la descàrrega s'ha completat:

```
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("Càrrega completada");
    }
```

O podríem recuperar informació ID3 d'un fitxer .mp3, com per exemple el nom de la cançó:

```
private function infoHandler(e:Event):void {
    trace("Nom de la cançó : " + sound1.id3.songName);
    }
```

2) Reproducció i control de la lectura (classe Sound i SoundChannel)

La reproducció i el control de la lectura del so es reparteix entre la classe Sound i la classe SoundChannel.

Iniciarem la lectura del so gràcies al mètode play(startTime) de la classe Sound, on startTime és el temps en mil·lisegons en què volem que comenci la lectura.

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

Cada so en Flash s'assigna a un SoundChannel, i es poden tenir diferents objectes SoundChannel simultàniament (seria com parlar de diferents pistes d'àudio que es llegeixen simultàniament). Si recuperem aquest objecte SoundChannel, podrem controlar el so.

Podem, per exemple, aturar la lectura:

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

O veure en quina posició som de la lectura, gràcies a la propietat *position*:

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

Nota

Si volguéssim fer un botó de pausa, seria com un botó stop, però desaríem en una variable la posició de lectura en què ens trobem.

3) Manipulació del so (volum i balanç amb la classe SoundTransform)

D'altra banda, podrem manipular el volum i balanç del canal de so. Per a això farem servir un objecte SoundTransform, que aplicarem al canal de so (SoundChannel).

Inicialment creem un objecte SoundTransform. El constructor de SoundTransform accepta uns valors inicials de volum i balanç:

```
var soundTransform1:SoundTransform = new soundTransform (volume,pan);
//On volume és un valor de 0 (silenci) a 1 (volum màxim)
//i pan, un valor de -1 (balanç en altaveu esquerre) a 1 (balanç en altaveu dret)
```

Una vegada creat l'objecte n'hi ha prou d'aplicar-lo al canal que vulguem. Si volem fer més canvis no caldrà crear un altre objecte SoundTransform, ja que podem modificar-ne les propietats i tornar a aplicar-lo al canal.

Vegem, per exemple, com podríem reduir el volum a la meitat i centrar el balanç:

```
//Modifiquem l'objecte SoundTransform
soundTransform1.volume = 0,5;
soundTransform1.pan = 0;

//ho apliquem al canal
soundChannel1.soundTransform = soundTransform1;
```

2.7.7. Vídeo

Per a carregar vídeos externs utilitzarem les classes Video, NetConnection i NetStream.

El procés es desenvolupa en quatre passos:

1) Creació d'un canal entre l'aplicació i la font

La classe NetConnection crea una connexió bidireccional entre Flash Player (o Air) i el servidor (o emplaçament del fitxer vídeo). Es podria dir que l'objecte NetConnection serveix de canal entre el client i el servidor.

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

Per a comprovar la connexió, escoltarem els esdeveniments NetStatusEvent.NET_STATUS i SecurityErrorEvent.SECURITY_ERROR de l'objecte NetConnection.

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

I utilitzarem el mètode connect() per a crear la connexió. En cas de connexions HTTP, o si el nostre vídeo és local, farem:

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

SecurityErrorEvent.SECURITY_ERROR, com el seu nom indica, ens informarà si hi ha problemes de seguretat en la connexió, mentre que NetStatusEvent.NET_STATUS ens informarà si la connexió s'ha realitzat i podem començar la descàrrega.

```
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) Creació d'una connexió de descàrrega

Una vegada comprovat el canal, utilitzarem NetStream per a obrir una connexió unidireccional de transmissió, que ens permetrà rebre el vídeo en temps real (*streaming*) i iniciar-ne la lectura encara que no s'hagi baixat íntegrament.

```
private function connectStream():void {

    // Una vegada comprovada la connexió, inicialitzem l'objecte NetStream,
    // al qual passem l'objecte NetConnection com a paràmetre
    // stream = new NetStream(connection);

    //Igual que amb NetConnection, podem escoltar els esdeveniments per a monitorar
    //l'estat de la connexió
    stream.addEventListener(NetStatusEvent.NET_STATUS, netStatusHandler);
    stream.addEventListener(AsyncErrorEvent.ASYNC_ERROR, asyncErrorHandler);

    // visualitzar stream de vídeo
   visualizeVideoStream();

}
```

3) Visualització de l'*stream* de vídeo

Ja tenim la connexió i recepció de l'*stream* de vídeo. Ara n'hi ha prou de fer servir la classe Video per a recuperar aquest *stream* i visualitzar-lo:

```
private function visualizeVideoStream():void {

    // Creem una instància de la classe Video
    video = new Video();

    // Li adjuntem l'stream vídeo
    video.attachNetStream(stream);

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

    // No s'ha d'oblidar afegir l'objecte Video a la DisplayList
    addChild(video);

}
```

4) Manipulació de l'*stream* de vídeo

Gràcies a la classe NetStream podem controlar la lectura del vídeo gràcies a mètodes com:

```
stream.stop();        // atura la lectura
stream.pause();       // pausa la lectura
stream.resume();      // reprèn la lectura
stream.seek(offset);  // ens permet desplaçar el capçal
                      // de lectura a la posició offset
                      // on offset és el nombre
                      // de segons des del principi
```

O propietats com:

```
stream.bytesLoaded;
stream.bytesTotal;   // amb els quals es pot visualitzar
                     // el percentatge ja carregat
stream.time;         // per a indicar-nos la posició del capçal
                     // de lectura (en segons)
```

Enllaç relacionat

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

També podem controlar el volum amb l'ús de la classe SoundTransform. El procediment és exactament igual al que hem vist amb el so, però ara l'aplicarem a la instància de NetStream.

Per exemple, per a reduir el volum del vídeo a la meitat faríem:

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

Enllaços relacionats

Un bon exemple:

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

Visualitzar vídeos de Youtube en una aplicació Flash gràcies a l'API Youtube:

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

2.8. Enriquiment de les nostres aplicacions

2.8.1. Mecanismes visuals: animació en línia de temps

Si deixem de banda el seu ús per a la creació d'aplicacions Rich Media, Flash Professional s'ha convertit en pocs anys en una eina ideal per a la creació d'animacions, i sovint és una alternativa de baix cost per a la creació d'animacions per a televisió. Fent servir la línia de temps, Flash ofereix la possibilitat d'animar mitjançant fotogrames clau (*keyframes*), fotograma a fotograma o amb interpolació de formes o de moviment. A més a més, disposa d'altres eines pràctiques, com ara *onion skins*, *bones*, etc., que són molt útils en animació.

El propòsit d'aquest capítol no és estendre's gaire sobre aquest tema, cosa que escaparia al propòsit del curs i ens endinsaria més en el món de l'animació que no pas en el de la programació en AS3, però sí que és recomanable tenir una idea del funcionament de les interpolacions de moviment i, després, veure els avantatges de fer-ho programàticament.

Vegem-ne el **funcionament** amb un exemple simple:

1) Per a començar i poder aplicar una interpolació de moviment haurem de crear un objecte en Flash Professional. Creem, per exemple, un cercle, el seleccionem i creem un objecte (F8). Des de Flash Professional solament optarem per gràfic o MovieClip.

2) Creem un fotograma clau (*keyframe*) d'inici (F8) en la línia de temps, suposem en el primer fotograma, i situem el nostre símbol cercle a l'esquerra de la pantalla.

3) Ara creem un altre fotograma clau (F8) més endavant en la nostra línia de temps (suposem el fotograma 10) i desplacem el cercle a la dreta de la pantalla.

4) Finalment, i tornant al primer fotograma, seleccionarem el nostre símbol i, fent clic amb el botó dret, escollirem interpolació clàssica.

Podem veure que, desplaçant-nos en la línia de temps, el cercle es mou de la part esquerra a la dreta.

Nota

També podem tenir altres formes més complexes si editem la corba de variació de la velocitat directament en Flash Professional.

5) La velocitat a la qual es desplaça el cercle del punt A al punt B és constant (interpolació per defecte). Però en el món real ben poques vegades tenim moviments a velocitat constant. Habitualment tindrem una acceleració constant, cosa que ens donarà una velocitat variable. Així, els objectes s'acceleraran o es desacceleraran progressivament, rarament ho faran a una velocitat constant.

Per a controlar aquesta evolució de la velocitat dins d'una interpolació, Flash Professional ens proporciona la propietat ease, que ens permetrà controlar com es produeix la interpolació.

ease = –100, indicarà una animació que comença lenta i va accelerant-se.

ease = 100, indicarà una animació que comença ràpida i progressivament va reduint la velocitat.

ease = 0, serà una animació lineal, a velocitat constant.

Resumint, podem veure que **una interpolació de moviment es basa en**:

- Una posició A inicial.
- Una posició B final.
- El temps en què passem de la posició A a la posició B (basat en el nombre de fotogrames de la línia de temps).
- Escollir com volem que evolucioni la velocitat entre aquests dos punts, el tipus d'*easing*.

2.8.2. Animació per programació: *tweens*

Acabem de veure com utilitzar Flash Professional per a obtenir animacions en la línia de temps. Però aquest tipus d'animació presenta un gran desavantatge. Són animacions fixes, estàtiques. Com podríem fer una animació interactiva?

Suposem, per exemple, que volem que l'animació anterior no vagi del nostre punt A al nostre punt B, sinó que volem que el cercle es desplaci a qualsevol punt de la pantalla en el qual fem clic amb el cursor del ratolí. És més, volem que si fem clic en un altre punt i encara no s'ha acabat el primer recorregut, el cercle rectifiqui la seva trajectòria per a desplaçar-se cap a aquest nou punt. Com ho faríem?

Veiem ràpidament que si ens cal certa interactivitat i canvis en les animacions, fer servir la línia de temps serà molt limitat; seria impossible crear totes les possibles animacions.

A més a més, fins ara hem parlat d'interpolacions de posició, però què passaria si volguéssim canviar una altra propietat (transparència, grandària, angle, etc.)? En aquest cas s'utilitzen els **paquets dedicats a crear interpolacions (*tweens*)**.

Cal destacar que encara que AS3 té la seva classe nativa *tween* pròpia, que permet crear aquestes interpolacions, hi ha **biblioteques de tercers**, que han demostrat, després de nombroses actualitzacions, ser molt més interessants i riques, amb un rendiment, una optimització de recursos i una rapidesa més alts i més possibilitats de control (tipus d'interpolacions, propietats que es poden interpolar, ús d'esdeveniments per a monitorar l'animació, etc.).

Entre les biblioteques més conegudes tenim **Tweener** (caurina), **TweenMax** (i la seva versió més reduïda **TweenLite**), **gTween** i les que puguin aparèixer en el futur.

Per a la resta del curs ens centrarem en **el paquet greensock** (TweenMax i TweenLite), un paquet molt complet. Tot i així, en general veureu que totes les biblioteques conegudes presenten característiques similars:

- Objecte al qual s'aplica la interpolació.
- Propietat o propietats que s'han de modificar.
- Valors inicials i finals.
- Durada de la interpolació (en fotogrames o mil·lisegons).
- Tipus d'interpolació (*easing*).
- Esdeveniments per a monitorar i manipular l'animació.

2.8.3. Iniciació a TweenMax

Abans de començar, haurem de descarregar el paquet amb les classes que necessitem: http://www.greensock.com/tweenmax/ i afegir-ho a la nostra carpeta /com de l'aplicació.

Vegem un **ús bàsic de TweenMax**:

1) Importem la classe TweenMax:

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

2) Creem una instància de la classe TweenMax que s'ocuparà de la interpolació:

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

On:

- **target**: és l'objecte al qual volem aplicar la interpolació.
- **duration**: és la durada en segons (mesura per defecte).
- **vars**: és un objecte amb les propietats que s'han de modificar.

Nota

Podem especificar la durada en fotogrames (*frames*) si canviem la propietat de la instància tween1.useFrames = true;

Així, si volem que l'objecte *balloon* es desplaci en un segon a la posició $x = 20$, $y = 30$ i, al mateix temps, que la seva opacitat canviï a la meitat, faríem:

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

Canviar el tipus d'interpolació:

Per a fer això, afegirem la propietat ease al constructor, cosa que es realitza per mitjà de l'objecte *vars*. Atenció: caldrà importar les classes per als diferents tipus d'*easing*.

```
import com.greensock.easing.*;

...

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

Si no ho especifiquem, TweenMax utilitzarà Quad.easeOut, però hi ha una gran varietat de tipus d'interpolació.

Detectar el final d'una interpolació:

Sovint ens interessarà també detectar el final d'una interpolació per a executar una acció concreta.

Per exemple, podríem voler que un objecte canviï d'opacitat progressivament fins a ser totalment transparent i, després, esborrar-lo de la DisplayList (i d'aquesta manera optimitzaríem l'ocupació dels recursos per part de l'aplicació).

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

Sovint, quan vulguem executar més d'una instrucció en finalitzar una interpolació, serà millor crear una funció separada que utilitzarem com a *callback* al final de la interpolació. Així obtindrem un codi més ben organitzat i les instruccions que cal executar no quedaran condensades dins de la pròpia instanciació de TweenMax.

Si reprenem l'exemple anterior, faríem simplement:

Enllaços relacionats

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

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

// Atenció, fixeu-vos que només posem el nom de la funció
// Sense afegir ()
// Si volem enviar paràmetres a aquesta funció
// farem servir l'Array onCompleteParams

...

private function elMeuMetode():void {
   //Instruccions que cal realitzar
   removeChild(balloon);
}
```

2.8.4. Emmagatzematge de dades persistent: *shared objects*

En la creació d'ARM sovint ens trobarem la necessitat de conservar dades en format local, directament en el dispositiu de l'usuari. Podem voler desar les preferències que l'usuari ha configurat, conservar l'estat de l'aplicació de manera que en iniciar-la de nou s'obri en el mateix estat en què estava quan es va aturar, desar dades de puntuació en un joc, etc.

Flash, mitjançant l'ús de **Shared Objects**, ens permet desar aquestes dades en local. Podríem dir que un Shared Object funcionarà de manera molt similar a les galetes (*cookies*) dels navegadors, amb la diferència que ara és Flash qui gestiona aquests fitxers.

El seu funcionament és bastant simple.

1) Creació d'una instància de la classe Shared Object:

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

Podeu veure que, a diferència d'allò que s'acostuma a fer en la creació d'instàncies, en aquest cas no utilitzarem new(). Per contra, farem servir directament el mètode getLocal(), que comprovarà que existeixi el Shared Object:

- Si el Shared Object no existeix (primera vegada que executem l'aplicació), el crearà i l'assignarà a la instància appSharedObject.

- Si el Shared Object ja existeix (ja s'ha executat una primera vegada l'aplicació i ja l'hem creat anteriorment), simplement el recuperarà i l'assignarà a appSharedObject.

2) Desar dades en el Shared Object:

Per a desar dades en un Shared Object, primer utilitzarem la propietat .data per tal de crear les diverses dades que vulguem conservar. Indicarem una variable i el seu valor corresponent; si la variable no existeix, la crearà, i si existeix, la sobreescriurà amb el seu valor nou.

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

Una vegada creades les diverses dades que vulguem conservar, farem servir el mètode flush() perquè es desin les dades en local:

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

3) Recuperar dades del Shared Object:

La recuperació de dades es realitza seguint pràcticament el mateix procés. Utilitzarem el mètode getLocal() per a recuperar l'objecte SharedObject de la nostra aplicació. Per a això necessitarem l'id, l'identificador que hem utilitzat inicialment:

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

Una vegada recuperat el SharedObject, podrem fer servir la propietat .data per a recuperar les dades que necessitem:

```
trace (appSharedObject.data.film);          ---> "Reservoir dogs"
trace (appSharedObject.data.director);      ---> "Quentin Tarantino"
```

4) Esborrar dades d'un Shared Object:

En la creació d'ARM no cal oblidar mai l'usuari i la seva privadesa. Podríem oferir la possibilitat d'activar o desactivar l'opció de desar informació en local o oferir la possibilitat d'esborrar aquestes informacions quan l'usuari ho vulgui.

En aquest cas, n'hi haurà prou de fer servir la funció clear().

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