

Conocimientos preliminares

Antes de comenzar cualquier proyecto de Flash, debemos conocer sus nociones fundamentales, tales como tipos de datos, operaciones, arrays y estructuras. El objetivo de este apartado tiene como función aclarar estos conceptos primordiales, que serán utilizados a lo largo del libro.

MovieClips	16
Génesis	16
Anatomía	17
Comunicación	18
Orden de apilamiento	19
Remoción	20
Variables	21
DataTypes	22
Operaciones con variables	24
Strings	25
Representación de strings	25
Numbers	27
Operaciones numéricas	27
Funciones matemáticas	28
Arrays	29
Estructuras de control	32
Funciones	38

MOVIECLIPS

Flash está basado en una estructura de películas: **todo** transcurre en un escenario (**stage**) regido bajo las leyes del tiempo. Cada actor que arrastramos al área de trabajo (texto, gráfico, fotografía, etc.) se posa sobre una línea de tiempo, a la par que toma su nueva condición de película.

En otras palabras, hay que pensar en Flash como una gran animación capaz de contener infinitas animaciones de menor calibre dentro de él; en un escenario repleto de actores con vida propia que pueden ser manipulados a través del ActionScript. Es justamente esta última condición de **películas manipuladas a través de código** que otorgan a Flash la etiqueta de **herramienta de interactividad**, y al MovieClip (película), en la columna vertebral de la aplicación.

Génesis

Un MovieClip puede ser generado de varias maneras:

- **Drag & Drop:** encapsulando el contenido gráfico del escenario y transformándolo como símbolo MC.
- **Réplica:** mediante la función `duplicateMovieClip()`, realiza una copia de un MC existente en el escenario.

```
// instancia_mc.duplicateMovieClip("nuevo_nombre", profundidad)

// suponiendo que cuento con "mc1" en el escenario
var mc1:MovieClip;
mc1.duplicateMovieClip("mc2",1);
```

- **Generación espontánea:** se da origen a un nuevo MC vacío, que nos permitirá incorporar contenidos en su interior.

```
// instancia_mc.createEmptyMovieClip("nuevo_nombre", profundidad)

// cuento con una imagen "archivo.jpg" en la misma ruta del .swf
this.createEmptyMovieClip("foto",1);
foto.loadMovie("archivo.jpg");
foto._rotation=70;
```

- **Linkage:** réplica de un MC no existente en el escenario.

```
// suponiendo que el identificador de linkage es "renton"
this.attachMovie("renton","nuevo_mc",1);
```

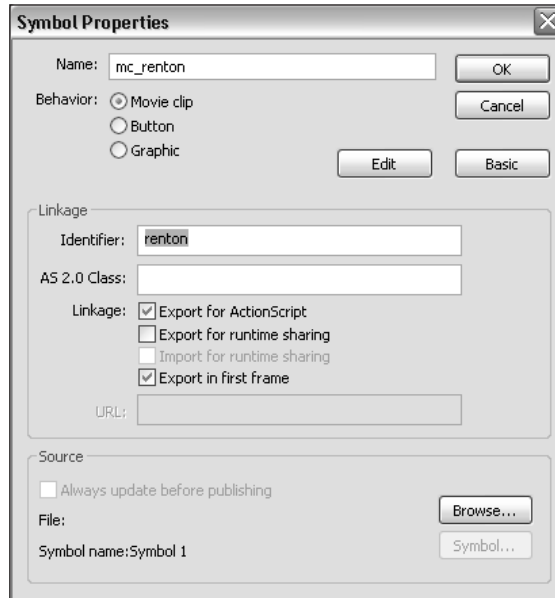


Figura 1. Cuando se le asocia un identificador a un MovieClip (linkage), éste puede ser instantáneamente referido sin necesidad de que exista físicamente en el escenario.

Anatomía

Una vez que un elemento ha sido situado en el escenario y convertido en MovieClip, adquiere una composición dada por tres partes fundamentales: nombre único, propiedades y métodos capaces de realizar.

- El **nombre de instancia** permite identificar al MovieClip para su posterior referenciación/comunicación a través del código.
- Las **propiedades** nos brindan la función de describir físicamente al objeto, permitiéndonos indicarle qué posición debe ocupar en el escenario (**x**, **y**), establecer sus dimensiones (**_height**, **_width**), su grado de rotación (**_rotation**) o su posición relativa al mouse (**_xmouse**, **_ymouse**).
- Los **métodos** son tareas específicas realizables por el MovieClip, entre las que encontramos **gotoAndStop()** (detenerse en cierto punto), **startDrag()** (desplazarse junto al mouse), o **duplicateMovieClip()** (clonarse).

Comunicación

Los lenguajes orientados a objetos (*OOP*, como lo es ActionScript) poseen una particular sintaxis de referenciación denominada **sintaxis de puntos**. Esto se debe ya que un punto (.) separa la referencia del objeto (instancia) de aquello que desea hacerse con él (manipular sus propiedades o métodos).

```
instancia_mc._rotation=90;
instancia_mc.nombre_variable="Macromedia";
instancia_mc.otra_instancia_mc.attachMovie("película.swf");
```

Como señalamos al comienzo, un MovieClip puede englobar dentro suyo textos, imágenes, sonidos, botones, variables... o bien otros MovieClips. Cada vez que incluimos un MovieClip dentro de otro, nos encontramos frente a una tarea de anidación, generando relaciones de parentesco entre dichos objetos involucrados: un MovieClip pasa a tener la condición de **hijo**, en tanto que un segundo MovieClip (ahora convertido en “padre”) lo encapsula.

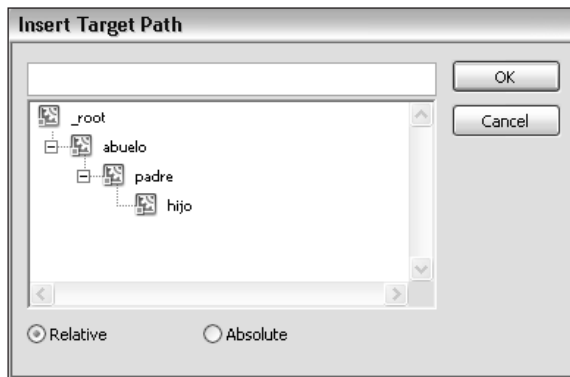


Figura 2. En este gráfico vemos la estructura de parentesco.

La estructura de anidación de MovieClips determina las relaciones de parentesco entre cada uno de dichos objetos (padre, hijo, hermano, abuelo...).



ORIGEN DEL MENSAJE EN UNA *COMUNICACIÓN ABSOLUTA*

Al utilizar este tipo de comunicación, no precisamos saber el lugar de emisión del mensaje; siempre partimos de un MovieClip existente y conocido por el resto de los MovieClips (**_root**). No existe manera que un desarrollo cobre vida si éste carece de un escenario.

Esta anidación nos da indicios que **para llegar de un punto a otro** en la estructura de comunicación; deberemos optar por distintos caminos por recorrer.

```
// posibles caminos para ejecutar el método "play()"
// partida: _root -- llegada: padre
abuelo.padre.play();
this.abuelo.padre.play();
_root.abuelo.padre.play();
_level0.abuelo.padre.play();

// partida: abuelo -- llegada: hijo
padre.hijo.play();
this.padre.hijo.play()
_root.abuelo.padre.hijo.play();
_level0.abuelo.padre.hijo.play();

// partida: hijo -- llegada: abuelo
_parent._parent.play();
this._parent._parent.play();
_root.abuelo.play
_level0.abuelo.play();
```

Existen dos **tipos de comunicación** que nos permiten **zurcar** a través del árbol de anidación y alcanzar nuestro destino: **relativa** y **absoluta**. La primera de ellas siempre toma como punto de partida el MovieClip desde donde se ejecuta el código (**this**), mientras que la segunda (**_root**, **_leveln**) parte desde el tronco/inicio del árbol y de allí comienzan a descender hasta el objeto de destino.

Orden de apilamiento

Flash está basado en un sistema de películas dispuestas en **capas** que permiten todo tipo de combinaciones de apilamiento/profundidad.

Esto determina que un MovieClip pueda estar por encima o debajo de otro, y cuál o cuáles de ellos responden a la suerte que les depara a sus superiores (anidación).

- Un **_level (capa)** está compuesto por una única línea de tiempo principal (**_root**), a la par que es capaz de alojar infinitos MovieClips.
- El **_level0 (cero)** es el más importante de todos los levels, ya que todos se posan sobre su superficie. Si éste es removido, la misma suerte sucede con todos los restantes.
- Cuanto mayor sea el nivel de apilamiento, más **arriba** se encontrará.

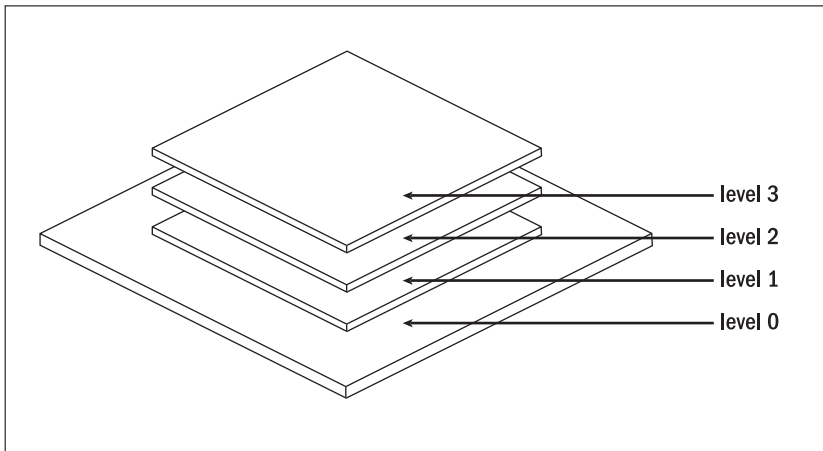


Figura 3. Orden de apilamiento de películas.

Remoción

De la misma manera que un MovieClip puede ser generado a través de código, también puede ser removido. La única condición necesaria para que esto ocurra es que haya sido creado por cualquier método que no sea el manual (es decir, mediante **createEmptyMovieClip()**, **duplicateMovieClip()** o bien **attachMovie()**).

```

/*
removeMovieClip()
para aquellas instancias de MC creadas mediante duplicateMovieClip(),
attachMovie() y createEmptyMovieClip()
*/
instancia_dinamica_mc.removeMovieClip();

/*
unloadMovie()
NO remueve al MC, tan sólo elimina sus contenidos.
Para aquellas instancias de MC creadas manual o dinámicamente
*/
instancia_mc.unloadMovie();

/*
unloadMovieNum()
remueve al nivel indicado entre paréntesis
*/
instancia_mc.unloadMovieNum(n);

```

VARIABLES

Una variable es un contenedor de datos, una porción de “recuerdos” grabados en la computadora que posteriormente podrán ser reinvocados y reutilizados de acuerdo con las necesidades del programador.

Se trata de una **caja etiquetada** capaz de alojar diferentes contenidos a lo largo de su vida. Si bien los elementos dentro de sí pueden alternarse, la referenciación a la misma permanece constante.

```

var texto;           // A) declaración variable
texto="Brainmotion"; // B) asignación inicial de contenido
texto=123;          // C) nuevo contenido
texto="macromedia"; // D) nuevo contenido
trace(texto);      // => macromedia (en ventana Output)
texto="";          // E) elimino datos; no el indicador
delete texto;      // F) elimino variable
trace(texto);      // => undefined (en ventana Output)

```

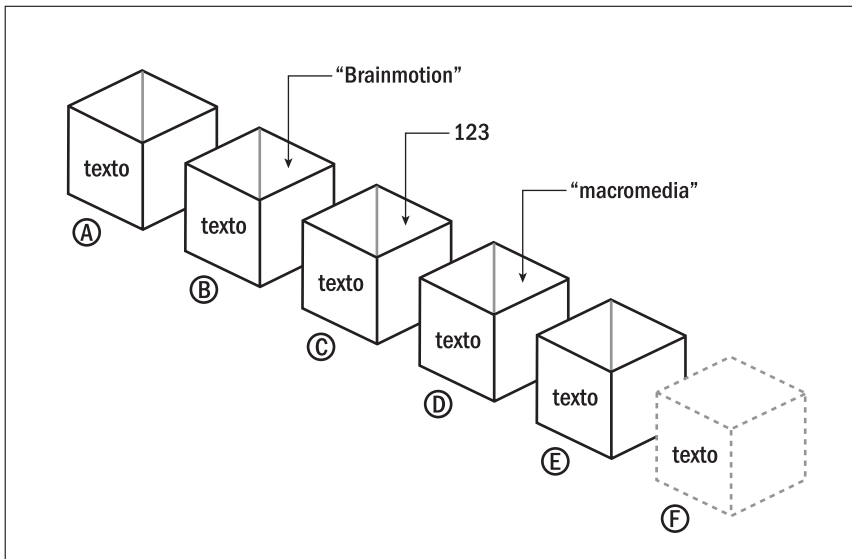


Figura 4. Posibles variaciones en los contenidos de una variable a lo largo de toda su vida.

DataTypes

Una variable es capaz de almacenar en su seno datos de diferente índole y características (texto, número, booleano...), capacidad que se ve determinada por el Data Type con el que se asocia a la misma en el momento de su génesis.

Principales DataTypes:

```

var nombre_variable:DataType
// declaracion
var a:String;
var b:Number;
var c:Boolean;
var d:MovieClip;
var e:Function;
var f:Object;
var g:Array;
var h:Color;
var i>Date;
var j:Sound;

// asignación
a="Mediarythmic";
b=12345;
c=true;
d=mi_mc; // referencia a un MovieClip "mi_mc"
function e(){};
f=new Object();
g=new Array();
h=new Color();
i=new Date();
j=new Sound();

```

Este **DataType** que acompaña a la variable determina la especificidad de los valores que ella puede almacenar, así como las operaciones que pueden hacerse con los mismos. En otras palabras: **determina qué datos pueden existir dentro del contenedor y cuáles no.**

III DECLARACIÓN DE VARIABLES

Si bien se aconseja asignar un `DataType` a una variable al inicializarse, esta tarea no es excluyente. En ambos casos, la variable **texto** contendrá el mismo valor y se comportará de igual modo.

```

var texto="texto";
var texto:String= "texto";

```

```
// (A) - ERROR !!
var texto:String;
texto="Brainmotion";
texto=123; // => ERROR
// (B) - ERROR !!
var musica:Sound;
musica="Brainmotion"; // => ERROR
```

En el caso **(A)**, la variable **texto** es designada como un String. Es por ello que ante la intención del programador de asignarle posteriormente otro tipo de datos a la misma (un valor numérico), el intérprete de Flash puntualiza que se ha realizado una operación errónea. Con distintos valores, lo mismo sucede en el caso **(B)**:

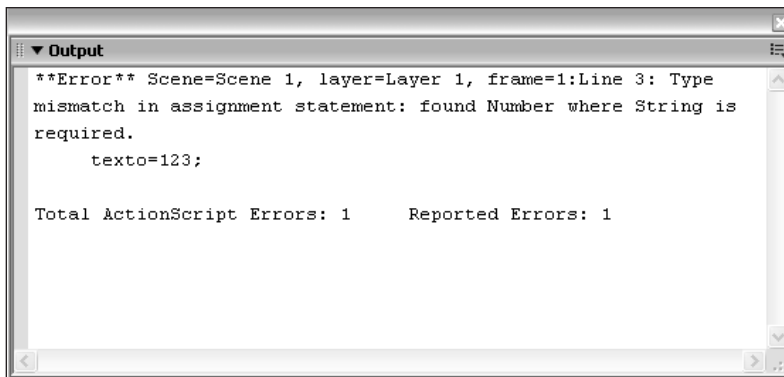


Figura 5. El intérprete de Flash se encarga de “señalar” (ventana de *Output*) que el dato asociado a una variable no corresponde al *DataType* predefinido.

La función **typeof()** permite consultar el tipo de dato existente en una expresión:

```
typeof ("macromedia"); // => string
typeof (98789); // => number
```

III CONVERSIÓN DE DATATYPES

Array(), **Boolean()**, **Number()**, **Object()** y **String()** nos permiten forzar la conversión de valores entre diferentes *DataTypes*;

```
123 + "456"; // => 123456
```

```
123 + Number ("456"); // => 579
```

Operaciones con variables

La operatoria de variables está determinada por las capacidades inherentes al valor o los valores que ésta transporta.

```
// --- String + String
var a:String, b:String
a= "Apple";
b="Macintosh";
trace (a + " " + b);           // => Apple Macintosh
trace (Apple "+" "Macintosh") // => Apple Macintosh

// ----- Number + Number
var a:Number, b:Number;
a= 12345;
b= 11111;
trace (a + b);                 // => 23456
trace (12345 + 11111);        // => 23456

// ----- String + Number
var a:String, b:Number, c:String;
a= "Agente";
b= 007;                         // número 7
c= "James Bond"
trace (a + " " + b + ": " + c); // => Agente 7: James Bond

// --- String + String (Number)
var a:String, b:String, c:String;
a= "Agente";
b= "007";                         // número 7
c= "James Bond"
trace (a + " " + b + ": " + c);   // => Agente 007: James Bond
trace ("Agente "+"007:" + " James Bond"); // => Agente 007: James Bond
```

STRINGS

Un **String** (texto) consta de una cadena de caracteres agrupados entre un par de comillas. Estas últimas pueden ser simples o dobles, pero no intercalarse.

```
"Libro de Actionscript"; // dobles
'Libro de Actionscript'; // simples
"Libro de Actionscript"; // ERROR !!!
```

Flash nos permite realizar diversas tareas predeterminadas con estos strings valiéndose del rico diccionario de métodos y propiedades que posee. Entre estas funcionalidades, podemos destacar las siguientes:

- **Búsqueda de caracteres específicos: indexOf(), lastIndexOf()**

```
var temp:String
temp="Firefox amenaza la supremacía de Explorer";
trace (temp.indexOf ("sup")); // => 19 (posicion de caracter)
trace (temp.indexOf ("Microsoft")); // => -1 (no encontrado)
```

- **Conversión a mayúscula/minúscula: toUpperCase(), toLowerCase()**

```
trace (temp.toUpperCase()); // => FIREFOX AMENAZA LA SUPREMACÍA ...
trace (temp.toLowerCase()); // => firefox amenaza la supremacía ...
```

- **Cantidad de caracteres en un String: length**

```
trace (temp.length); // >> 41
```

Representación de Strings

En Flash existen dos grupos de contenedores visuales capaces de representar gráficamente un String: los Campos de Texto Nativos (con sus variantes Estático, Dinámico y de Input) y los Componentes de Texto (dentro de los que encontramos a Etiqueta, Área de Texto e Input de Texto).

Cada uno de ellos poseen funcionalidades diferentes, lo que determina sus distintos roles en una película Flash:

- **Campo Estático (*static*):** utilizado para aquellos contenidos que únicamente cumplen funciones gráficas, por lo que no precisan ser actualizados, por ejemplo, palabras sueltas, párrafos o titulares, nombres de marcas, etc.

- **Campo Dinámico (*dynamic*):** utilizado cuando necesitamos operar con información cuya representación gráfica pueda ser actualizable en respuesta a diferentes situaciones (contenido del campo controlado a través de ActionScript).
- **Campo de Input (*input*):** es muy similar al Dinámico, con la inclusión de una función que permite capturar aquella información escrita por el usuario.

El grupo de componentes de texto de la nueva versión MX 2004 responde a la necesidad de acompañar la nueva arquitectura funcional de dicha versión. En cuanto a lo que concierne la representación directa de texto, este conjunto no ofrece mayores aportes que los Campos de Texto Nativos.

- **Etiqueta (*label*):** ingreso de caracteres en pequeña cantidad y una única línea.
- **Área de Texto (*textArea*):** ingreso de caracteres que construyen oraciones y párrafos. En cuanto a la información ingresada sobrepasa la superficie visible de caracteres, automáticamente se dibuja una barra de scroll en el margen derecho.
- **Input de Texto (*textInput*):** captura de datos del usuario.

La lectura y escritura de datos es igual para todos los contenedores de texto: ActionScript utiliza la propiedad **text**.

```
// Campos de texto nativos
instancia_dynamic.text="Campo de Texto Dinámico";
instancia_input.text="Campo de Texto de Input";

// Componentes de texto
instancia_label.text="Componente Label";
instancia_textarea.text="Componente Text Area";
instancia_textinput.text="Componente Text Input";
```

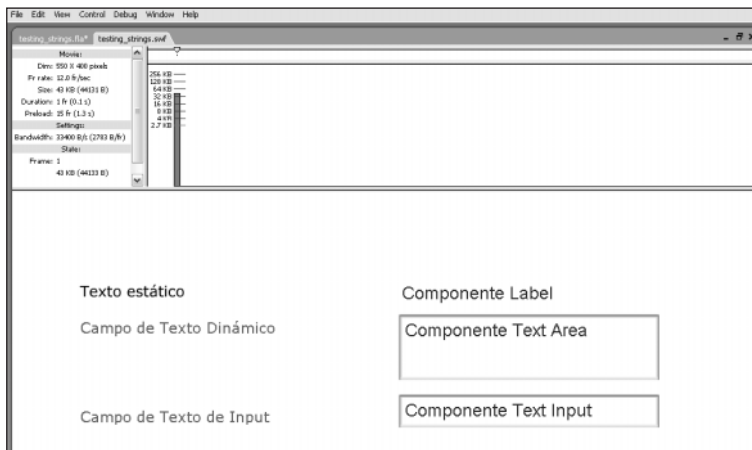


Figura 6. Los 6 tipos de campos de texto de Flash: Campos de Texto Nativos (Estático, Dinámico y de Input) y Componentes de Texto (Etiqueta, Área de Texto e Input de Texto).

NUMBERS

Los datos del tipo **Number** (*número*) pueden estar comprendidos dentro del universo de los enteros, decimales o especiales.

```
var num:Number;
// número entero
num=1979;
num=0;
num=-1979;
// número decimal
num=19.79;
num=-19.79;
num=19.0;
// número especial
250/0;           // Infinito
```

Operaciones numéricas

Los números pueden ser manipulados a través de los operadores aritméticos de adición (+), sustracción (-), multiplicación (*), división (/) y resto o módulo (%).

Toda operación matemática en Flash sigue las mismas reglas de orden que la matemática tradicional, como ser:

- Los signos “+” y “-” separan términos.
- Los paréntesis agrupan tareas de evaluación.

```
trace (2 + 5 * 10 - 3);           // => 49
trace ((2 + 5) * 10 - 3);       // => 67
```



NEWLINE

newline nos permite incluir un salto de línea en un string, tal cual sucede al tipear el código de escape `\n`.

```
trace ("línea 1\nlínea 2");
trace ("línea 1" + newline + "línea 2");
```

Existen otros dos operadores de incremento (++) y decremento (--) que facilitan la tarea de adicionar o sustraer en un uno (1) un valor determinado: **valor++**, **++valor**, **valor--** y **--valor**. Las siguientes expresiones producen los mismos resultados:

```
valor = valor+1;
valor+=1;
valor++;
```

Módulo (%)

El módulo entre dos cifras representa el resto surgido de dividir el primer operando con el segundo:

```
trace (14 % 4) ; // => 2 (resto)
```

14 puede ser dividido entre 3 partes iguales, resultando 2 como resto.

$$14 = 4 \times 3 (+ 2)$$

La representación que puede hacer Flash de un número se limita a 15 caracteres. Esto nada tiene que ver con el almacenamiento o manejo de los mismos, tan sólo con su “renderización”.

```
var n:Number=1234567890.1234567890;
trace(n) // >> 1234567890.12346
trace (10/3) // >> 3.333333333333333
```

Funciones matemáticas

Más allá de las operaciones tradicionales capaces de ser realizadas con Flash (suma, resta, multiplicación, división y módulo), ActionScript cuenta con una clase nativa denominada **Math**, donde se encuentra el mayor repertorio de operaciones con números:

- **Máximo y mínimo: min(), max()**

```
Math.max (9,2) // => 9 (valor más alto de los dos parámetros)
Math.min (9,2) // => 2 (valor más bajo de los dos parámetros)
```

- **Redondeo de valores: ceil(), floor(), round()**

```
Math.ceil (5.42);           // => 6 (entero próximo hacia arriba)- cielo
Math.floor (5.42);         // => 5 (entero próximo hacia abajo) - piso
Math.round (5.42);         // => 5 (entero próximo)- redondeo
```

- **Valor absoluto: abs()**

```
Math.abs (2.4);            // => 2.4 (valor despojado de signo + o -)
Math.abs (-2.4);           // => 2.4 (valor despojado de signo + o -)
```

- **Raíz y exponente: sqrt(), pow()**

```
Math.sqrt(9)               // => 3 (raíz cuadrada de nueve:  $\sqrt{9}$ )
Math.pow(2,3)              // => 8 (dos al cubo:  $2^3$ )
Math.pow(4,2)              // => 16 (cuatro al cuadrado:  $4^2$ )
```

- **Operaciones trigonométricas: acos(), asin(), atan2(), cos(), sin(), tan()**

```
// resultado en radianes
Math.sin(45);              // => 0.850903524534118
Math.cos(45);              // => 0.52532198881773
Math.tan(45);              // => 1.61977519054386
```

Si bien Flash maneja los ángulos en radianes, nosotros estamos acostumbrados a su lectura en grados. Es por ello que debemos realizar la conversión de sistemas para comprender la operatoria:

```
grados= (radianes / Pi ) * 180
```

ARRAYS

Un **Array** (también conocido como **lista** o **arreglo**) es un contenedor de múltiples datos donde cada valor ocupa un lugar específico en la estructura.

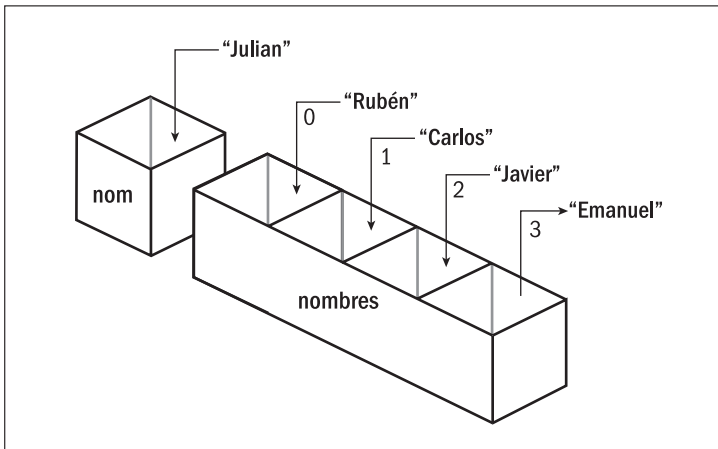


Figura 7. Visualmente podemos asociar un array con una caja. Cada sección recibe un índice de posición, que a su vez permite referenciar al elemento ocupado en dicho lugar.

El primer ítem es reconocido con el índice 0 (cero), el segundo con el 1 (uno), etc.:

```
var nom:String;
var nombres:Array;

nom="Julián";
nombres=["Ruben", "Carlos", "Javier", "Emanuel"];

/* análisis del Array ---
elem 0: "Ruben" (String)
elem 1: "Carlos" (String)
elem 2: "Javier" (String)
elem 3: "Emanuel" (String)
*/
```

Todo elemento que se incorpora a un Array recibe, ya sea de forma manual o automática, un **índice de posición**.

Génesis de Arrays

Todas las siguientes expresiones generan un Array **variable** con los mismos elementos:

```
var variable:Array
// opcion 1
variable = new Array (elem1, elem2, elem3);
```

```
// opción 2
variable = [elem1, elem2, elem3];

// opción 3
variable = [];           // o bien variable = new Array();
variable [0] = elem1;
variable [1] = elem2;
variable [2] = elem3;
```

Operaciones con Arrays

- Cantidad de elementos: `length`

```
var nombres:Array = ["Ruben", "Carlos", "Javier", "Emanuel"];
trace (nombres.length)    // => 4
```

- Incorporación de elementos: `push()`, `unshift ()`, `splice()`

```
var nombres:Array = ["Ruben", "Carlos", "Javier", "Emanuel"];
// PUSH: nuevo elemento al final del Array
nombres.push("Juan");
// => Ruben, Carlos, Javier, Emanuel, Juan

// UNSHIFT: nuevo elemento al comienzo del Array
nombres.unshift ("Máximo");
// => Máximo, Ruben, Carlos, Javier, Emanuel, Juan
// SPLICE: nuevo elemento en posición determinada
nombres.splice(2,0,"Julián");
// => Máximo, Ruben, Julián, Carlos, Javier, Emanuel, Juan
```

- Remoción de elementos: `pop()`, `shift ()`, `splice()`

```
var nombres:Array = ["Ruben", "Carlos", "Javier", "Emanuel"];
// POP: elimina último elemento del Array
nombres.pop();
// => Ruben, Carlos, Javier
// SHIFT: elimina primer elemento del Array
```

```
nombres.shift();  
// => Carlos, Javier  
  
// SPLICE: elimina elemento en posición determinada (desde, hasta)  
nombres.splice(1,1);  
// => Carlos
```

• Orden de elementos: reverse(), sort()

```
var nombres:Array = ["Ruben", "Carlos", "Javier", "Emanuel"];  
  
// SORT: orden alfanumerico ascendente  
nombres.sort();  
// => Carlos, Emanuel, Javier, Ruben  
  
// REVERSE: orden alfanumerico descendente  
nombres.reverse();  
// => Ruben, Javier, Emanuel, Carlos
```

ESTRUCTURAS DE CONTROL

Son construcciones específicas de código capaces de modificar el flujo normal del script. Entiéndase por **normal** a la linealidad de la ejecución de sentencias; al hecho de que en forma predeterminada Flash recorre las líneas de código una a una, de izquierda a derecha, y de arriba hacia abajo, a la par que las va ejecutando.

```
// flujo normal  
trace ("Veronica");  
trace ("Diego");  
trace ("Anton");  
// => resultado --  
Verónica  
Diego  
Anton  
// _____
```

Estructuras condicionales (if, if-else, switch-case)

Ejecutan una expresión/grupo de expresiones acorde con el resultado booleano (**true/false** o **verdadero/falso**) surgido de un determinado análisis.

- **if** (si)

```
// Teoría ---
if (true){
    // acción es ejecutada
}

// Práctica ---
if (6<5){           // FALSE
    trace ("6 es menor que 5");
}
// => (nada es devuelto ya que la condición "6<5" evalúa FALSA)
```

- **if - if** (si - y si)

```
// Teoría ---
if (true){
    if (true){
        // acción es ejecutada cuando ambas condiciones se cumplen
    }
}

// Práctica ---
if (2<5){           // TRUE
    if (2<4){
        trace ("2 es menor que 5 y que 4");
    }
}
// => 2 es menor que 5 y que 4
```

- **if - else** (si - entonces)

```
// Teoría ---
if (false){
```

```

        // acción no es ejecutada

    } else {
        // acción es ejecutada por descarte
    }
// Práctica ---
if (6<5){           // FALSE
    trace ("6 es menor que 5");
} else {
    trace ("FALSO: 6 no es menor que 5");
}
// => FALSO:6 no es menor que 5

```

- **if - else if** (si - en cambio si)

```

// Teoría ---
if (false){
    // acción no es ejecutada
} else if (true){
    // acción es ejecutada tras segundo chequeo
}

```

```

// Práctica ---
if (6<5){           // FALSE
    trace ("6 es menor que 5");
} else if (6<2){   // FALSE
    trace ("6 no es menor que 2");
}
// => nada es devuelto ya que ambas condicoines evalúan FALSAS

```

- **switch - case** (en caso de)

Es una variante de la estructura **if-else if**; de rápida escritura y constante evaluación de una misma condición.

```

// Teoría ---
switch (condicion){
    case true:

```

```

        // acción ejecutada
        break;    // se abandona la estructura
    case true:
        // acción ejecutada
        break;    // se abandona la estructura
    default:
        // acción ejecutada como último recurso
}

// Práctica ---
var temp:String="Julián";
switch (temp){
    case "Emanuel":    // FALSE
        trace ("El nombre es Emanuel");
        break;
    case "Julián":    // TRUE
        trace ("El nombre es Julián");
        break;        // abandona inmediatamente la estructura
    case "Juan":
        trace ("El nombre es Juan");
        break;
    default:
        trace ("No tiene nombre");    // nunca será ejecutada
}

// => El nombre es Julián

```

Estructuras iterativas o repetitivas (for, while, do-while)

Ejecutan un código incesantemente mientras se cumpla una condición booleana (**true**).

- **while** (mientras que)

```

// Teoria ---
while (condicion) {
    // acción ejecutada incesantemente
}

// Práctica ---
var tmp:Number=1;

```

```

while (tmp <= 3){
    trace ("Vez número: " + tmp);
    tmp++;      // incremento en 1 el valor de "tmp" con cada recorrido
}
// => Vez número 1
      Vez número 2
      Vez número 3

```

De no haber aumentado en 1 el valor de la variable **tmp**, la condición **tmp<=3** hubiese resultado siempre **verdadera**, por lo que el loop (**repetición**) jamás habría cesado. Esto se llama **loop infinito**, que inevitablemente, produce el “cuelgue” de Flash. Cuando el intérprete debe ejecutar la operación número 200.001 dentro de un loop, o bien cuando se encuentra procesando una simple instrucción por más de 15 segundos, el sistema emite un alerta que permite cancelar la ejecución (**Figura 8**).

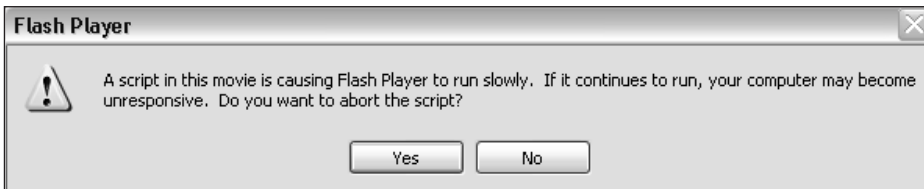


Figura 8. Mensaje que alerta sobre el “cuelgue” de Flash.

Cuando el intérprete ejecuta un script que tiende al **infinito**, Flash automáticamente advierte al programador de este presunto error en el código.

- **do - while** (haz - mientras que)

```

// Teoría ---
do {
    // acción ejecutada incesantemente
} while(condicion);

// Práctica —
var tmp:Number=1;
do {
    trace ("Vez número: " + tmp);
    tmp++;
} while (tmp <= 3);

```

```
// => Vez número 1
    Vez número 2
    Vez número 3
```

while es prácticamente igual a su par **do-while**, con la única diferencia que, en el último caso, el código dentro de las llaves del **do** siempre es ejecutado al menos una vez por más que la condición evaluada jamás sea verdadera.

- **for** (versión compacta de *while*)

```
// Teoría —
for (inicializador; condicion; actualizador){
    // acciones ejecutadas
}

// Práctica —
for (var tmp:Number=1; tmp<=3; tmp++){
    trace ("Vez número: " + tmp);
}

// => Vez número 1
    Vez número 2
    Vez número 3
```

Operadores lógicos y condicionales

Existen **operadores** que permiten evaluar una o varias expresiones dentro de un **if**.

OPERADOR	ACCIÓN	CATEGORÍA
&&	AND (y)	Lógico
	OR (o)	Lógico
!	NOT (no)	Lógico
==	igualdad	Condicional de igualdad
!=	desigualdad	Condicional de igualdad
>	mayor	Condicional de comparación
<	menor	Condicional de comparación
>=	mayor igual	Condicional de comparación
<=	menor igual	Condicional de comparación

Tabla 1. Operadores lógicos y condicionales.

```
// &&(AND)
if (condicion_1 && condicion_2 && condicion_3){
    // ejecutar si todas las condiciones se cumplan
}
// ||(OR)
if (condicion_1 || condicion_2){
    // ejecutar si cualquiera de las condiciones se cumplen
}
// !(OR)
if (!condicion){
    // ejecutar si la condicion NO se cumple
}
```

FUNCIONES

Una **función** consiste de una serie de instrucciones agrupadas bajo un único denominador. Cuando invoco el nombre de una función, se ejecutan todas las expresiones existentes dentro de la misma.

```
// Teoría —
var nombre_funcion:Function;
function nombre_funcion (){
    // ejecución de script
}

// Práctica —
// declaración de la función
var ejecutar:Function;
function ejecutar (){
    trace ("Se ha ejecutado la función");
}

// invoco la función
ejecutar();
// => Se ha ejecutado la función
```

Personalización de funciones

Las funciones tienen la capacidad de poder recibir **parámetros**, los cuales se encargan de individualizar la ejecución del código interno. En otras palabras, los **parámetros** nos permite reutilizar el script de una función, ejecutando nuevas y diferentes tareas con cada nueva invocación.

```
// Teoría —
var nombre_funcion:Function;

function nombre_funcion (param1:DataType, ...){
    // ejecución de script
}

// Práctica —

// declaración de la función
var saludar:Function;

function saludar (nombre:String){
    trace ("Buenos días " + nombre);
}

// invoco la función con un parámetro "Verónica"

saludar ("Verónica");

// => Buenos días Verónica

saludar ("Natalia");
// => Buenos días Natalia
```

El intérprete de Flash reemplaza los parámetros declarados entre paréntesis por su correspondiente valor en el momento de ejecución.

Interrupción de ejecución

El comando **return** termina abruptamente con el procesamiento de una función, evitando que toda línea de código posterior sea siquiera leída.

```
// declaración de la función
var listar:Function;
```

```
function listar (){
    trace ("Madre");
    trace ("Padre");
    return; // fin de ejecución
    trace ("Hermano");
}
listar();

// => Madre
      Padre
```

Cuando **return** es precedido por otra expresión, su valor es devuelto a la par que la función es abandonada.

```
// -- promedio de 3 notas --
var promedio:Function;

function promedio (nota1:Number, nota2:Number):Number{
    var promedio = (nota1+nota2) /2;
    return promedio;
}

var prom_juan:Number=promedio(10,7);
trace (prom_juan);
// =>      8,5

var prom_vero:Number=promedio(5,4);
trace (prom_vero);
// =>      4,5

var prom_mati:Number=promedio(6,10);
trace (prom_mati);
// =>      8
```