mosaic
comunicación interactiva
y tecnologías multimedia

Graduado en Multimedia

Carlos Casado Martínez

```
4  function crearNieve() {
5      i = _root.getNextHighestDepth();
6      tmp = _root.attachMovie("copo", "copo
7      tmp._x = randRange(0, 600);
8      tmp._y = -1;
9      tmp._alpha = randRange(50, 100);
10     tmp.speed = randRange(2, 7);
11     tmp._xscale = randRange(70, 110);
12     tmp._yscale = tmp._xscale;
13     tmp.moving = true;
14     tmp.onEnterFrame = moverNieve;
15 }
16 function moverNieve() {
17     if (this.moving) {
```

Introducción a la programación

Aprendiendo a programar con ActionScript

http://mosaic.uoc.edu

Índice

Índice 2
Introducción 4
Programas y datos 5
  Programas 5
  Datos 6
  Resumen y actividades 6
    Actividades 6
Algoritmos y programas 7
  Lenguajes de programación 7
  Algoritmos 9
  Resumen 9
Datos 10
  Datos, tipos de datos y variables 10
    Variables booleanas 12
  Asignación, lectura y escritura de datos 13
  Operadores y operaciones 14
    Operaciones matemáticas 14
    Autoincremento y autodecremento 15
    Operaciones con cadenas 15
  Comparaciones 16
    La comparación como operación 16
  Prioridad de las operaciones 17
  Operadores lógicos o booleanos 18
    Y lógico 18
    O lógico 18
    No lógico 20
  Resumen y actividades 20
    Resumen 20
    Actividades 20
Estructuras de control sencillas 22
  Introducción 22
  Estructura secuencial 22
  Estructura condicional sencilla 23
  Estructura condicional doble 24
  Resumen 25
  Actividades 25

Estructuras de control complejas (bucles) 26
- Introducción 26
- Para 26
- Mientras 27
- Resumen 30
- Actividades 30

Introducción al ActionScript 31
- Introducción 31
- Tipos de datos y operaciones 31
- If else 32
- for, while 32
- Actividades 33

Otras estructuras 37
- Introducción 37
- Búsqueda y recorrido 37
  - Recorrido 37
  - Búsqueda 38
- Subprogramas: Funciones y procedimientos 39
  - Parámetros 39
  - Paso de parámetros: Paso por valor y paso por referencia 40
  - ¿Qué utilidad tienen los diferentes pasos de parámetros? 41
- Variables en Flash 41
  - Variables, variables locales y variables globales 41
- Funciones en Flash 42
- Objetos, métodos y propiedades en Flash 42
  - El tipo Object 43
- Ejercicios 44
  - Búsqueda y recorrido 44
  - Funciones 44
  - Programación 44
  - Ejemplos 45

Introducción

Aprender a programar bien ha sido siempre una tarea ardua. Aunque son muchas las personas que de manera autodidacta han aprendido lo suficiente como para hacer sus propios programas, la verdad es que, como en todo, sin una buena base, programar bien no es trivial.

Sentarse delante de un ordenador a escribir un programa informático es una tarea que requiere además de una buena dosis de paciencia, unos conocimientos básicos que permitan al programador tener una cierta seguridad de que lo que está haciendo funcionará sin demasiados problemas. Tener que probar lo que haces a cada paso representa una pérdida considerable de tiempo, así que si tienes claro todo lo que puedes o no hacer y como hacerlo, siempre estarás en disposición de mejorar la calidad de tus programas.

Este manual pretende ser una breve introducción a la programación. Está pensado básicamente para que sea útil a aquellas personas que desean empezar a programar en ActionScript, el lenguaje de programación del Flash de Adobe (antes Macromedia). Sin embargo también puede ser de gran ayuda para aquellas personas que quieran aprender a programar, casi, para cualquier lenguaje de programación.

Con este objetivo, una parte importante del manual utiliza un pseudolenguaje de programación propio que no puede compilarse ni interpretarse. La idea es que el lector se entretenga en escribir sobre “el papel” sus programas; más adelante ya tendrá tiempo de hacerlo directamente sobre la máquina.

Este pseudolenguaje está pensado para que el salto desde él hasta ActionScript (el lenguaje de programación de Flash), sea muy sencillo. Eso complicará la existencia a aquellos que quieran practicar con otros lenguajes de programación, pero menos de lo que pudiese parecer.

Acabar con una advertencia. Este manual es una introducción. Seguir los pasos que aquí se indican facilitará a lector el aprendizaje de técnicas más avanzadas de programación. Eso sí, hay que tener paciencia y tesón.

Programas y datos

Programas

Al leer el título de esta etapa (Programas y datos) es fácil pensar en ordenadores y en una persona sentada delante de una pantalla durante horas, haciendo algo parecido al navegador de Internet que estamos usando para leer este texto. Sin duda este material tratará el tema de los programas desde un punto de vista similar, pero, para entender qué es un programa, debemos ser conscientes de que a lo largo de nuestra vida son muchas las ocasiones en las que nos dedicamos a programar muchas cosas que no son ordenadores.

Cuando nos vamos de viaje programamos el viaje. Cuando queremos que el vídeo nos grabe una película de la televisión programamos el vídeo. Si tenemos que dar unas clases tenemos que preparar el programa que daremos.

A veces el programa ya viene hecho y nosotros sólo tenemos que utilizarlo. Y no estamos hablando de programas informáticos. La lavadora, el lavavajillas, muchos electrodomésticos tienen un conjunto de programas que nosotros debemos escoger para que hagan su función. Podemos encontrar programas por todas partes sin necesidad de tener delante un ordenador. ¿Qué es entonces un programa?

Podríamos definir un programa como un conjunto de instrucciones que nos permiten, dada una situación inicial, llegar a la situación final que nosotros deseemos. Por ejemplo: el programa de ropa delicada de la lavadora nos permite dada una situación inicial (un conjunto de prendas delicadas y sucias) llegar a una situación final que nosotros deseamos (esas prendas limpias y sin estropear). Cada uno de los programas de los que consta la lavadora nos permite partir de una situación inicial diferente para llegar a una situación final similar.

El ejemplo de la lavadora nos ha mostrado como diariamente trabajamos con programas sin darnos cuenta. Pero, ¿y programar? Programar también lo hacemos auque seguramente no tan a menudo. Vamos a ver otro ejemplo: cuando queremos que el vídeo nos grabe una película de la televisión debemos programarlo. ¿En qué consiste programar el vídeo? La verdad es que es muy sencillo. Le indicamos qué cadena tiene que grabar, a qué hora debe iniciar la grabación y a qué hora debe finalizarla. Así, si queremos que grabe un programa de TVE 2, que empieza a las 15:30 y calculamos que acabará sobre las 17:00, al vídeo le daremos estas instrucciones:

- A las 15:30 horas ponte en marcha.
- Selecciona la cadena TVE 2.
- Inicia la grabación.
- A las 17:00 detén la grabación.
- Apágate.

Nuestro programa tiene 5 instrucciones que, si bien no las hemos escrito exactamente así, sí que hemos hecho lo suficiente como para que el vídeo las realice de esa manera. En este programa la situación inicial era una cinta de vídeo vacía y dentro del vídeo, y la situación final la misma cinta, en el mismo sitio, pero con el programa que queríamos ver grabado en ella.

Aunque parezca que el ejemplo del vídeo está muy lejos de un programa informático, no es así. Un programa informático no es más que un conjunto de instrucciones que el ordenador obedece ciegamente. Sin duda, el lenguaje que utilizaremos para indicar al ordenador que debe hacer es más complejo que el que hemos usado aquí, pero la base es la misma. En el programa informático partiremos siempre de una situación (estado) inicial y llegaremos a una situación (estado) final después de realizar un conjunto de instrucciones.

Datos

Hasta aquí hemos hablado de cómo podemos hacer un programa que haga un determinado trabajo y hemos intuido que hay mucha similitud entre un programa informático y un programa que nos permita hacer algo con un electrodoméstico.

Sin embargo, cuando hacemos un programa informático, este trabaja con unos datos (llamamos datos a cualquier información que introducimos en el ordenador), algo que no pasa con el vídeo o la lavadora. ¿O sí? ¿Con qué trabaja la lavadora? Evidentemente con ropa sucia. De alguna manera si hacemos un símil entre una lavadora y un ordenador, podríamos decir que la lavadora tiene un programa que, a partir de unos datos introducidos (ropa sucia), nos devuelve unos datos tratados (ropa limpia). Puede parecer difícil ver la ropa como un dato, pero fijémonos ahora en el ejemplo del vídeo. En este caso tenemos un programa que nos permite, a partir de unos datos de entrada (hora de inicio, hora de final, cadena y los datos de la antena de televisión), obtener unos datos de salida (un programa en una cinta de vídeo).

Así pues podríamos decir que cualquier programa informático (y, generalizando, cualquier programa) realiza un conjunto de instrucciones de manera que, a partir de una situación y de unos datos iniciales, llegar a una situación y a unos datos finales.

Resumen y actividades

Para resumir vamos a definir algunos de los términos que hemos utilizado y que seguiremos usando a lo largo de este documento.

Programa

> Conjunto de instrucciones que indica a un ordenador que tiene que hacer para, a partir de un estado inicial, llegar a un estado final.

Estado inicial

> Situación en que se encuentran el ordenador y los datos antes de ejecutar el programa.

Estado final

> Situación en que se encuentran el ordenador y los datos después de ejecutar el programa.

Dato

> Cualquier información que proporcionemos al ordenador.

Actividades

1. Escribe cuáles son las instrucciones que realiza el programa de lavado suave de tu lavadora (cargar agua, cargar jabón, …)
2. Escribe un programa que permita a una persona desplazarse desde tu domicilio hasta el ayuntamiento de tu localidad. ¿Qué instrucciones has utilizado? Escríbelas.
3. Escribe un programa que permita a un robot que sólo sabe interpretar las siguientes instrucciones: ir a comedor, coger silla, dejar silla, ir a dormitorio; mover seis sillas del comedor al dormitorio.
4. Escribe un programa que permita a una persona hacer una tortilla de patatas para dos personas.
5. Modifica el programa que has hecho para la actividad anterior de manera que permita a una persona hacer una tortilla de patatas para 2, 4 o 6 personas según el número de invitados que tenga.

Algoritmos y programas

Lenguajes de programación

Cuando debemos hacer un programa para un ordenador no podemos escribirlo de cualquier manera. Una de las actividades de la etapa anterior consistía en hacer un programa que permitiese a un robot mover unas sillas de una habitación a otra. Ese ejercicio especificaba que el robot sólo sabía interpretar cuatro instrucciones y por tanto para poder “explicarle” que tenía que hacer debíamos hacerlo sólo con ellas.

Cuando hacemos un programa para un ordenador nos encontramos en una situación similar: no podemos explicarle con nuestras propias palabras qué tiene que hacer. El ordenador es una máquina electrónica que funciona a base de impulsos eléctricos de manera que, según pase o no electricidad por unos circuitos determinados, hace unas cosas u otras. No vamos a entrar aquí en el sistema que utiliza el ordenador para trabajar, pero lo que es importante que comprendamos es que no nos podemos comunicar con él en su propio lenguaje (de impulsos eléctricos), ni en el nuestro. Por tanto, necesitamos algún lenguaje que esté a medio camino entre lo que entiende el ordenador y lo que podemos explicar nosotros.

Desde hace algún tiempo existen investigaciones tendentes a buscar una forma de comunicación con los ordenadores lo más cercana posible al hombre. Sin embargo todavía está lejos el día en que podamos hablar con los ordenadores y ellos nos entiendan (a pesar de “2001: Una odisea del espacio”). Así pues, nos tenemos que conformar con lo que llamamos lenguajes de programación, que no son más que “idiomas” que, simplificando el inglés, intentan facilitar la creación de programas.

Cuando una persona quiere crear un programa lo primero que debe hacer es planificar la tarea a realizar, analizar los datos de que dispone, saber exactamente cuál es el resultado a obtener. En la asignatura Programación: Programas y sistemas de autor I se explica con exactitud los pasos a dar para la creación de un programa. Como el objetivo de este documento es introducir al lector en la programación, nos limitaremos a hacer pequeños programas fáciles de planificar y dejaremos para dicha asignatura el saber cuáles son los pasos necesarios parar crear un programa complejo.

Lenguajes de programación hay muchos, pero nosotros nos centraremos en un lenguaje en particular: ActionScript, un lenguaje que nos permitirá hacer que Flash se convierta en algo más que un programa de animación. El ActionScript facilita que nuestras animaciones puedan interactuar con el usuario.

A pesar de que nuestro objetivo es aprender a programar en ActionScript, lo cierto es que no vamos a empezar directamente con él. Primero veremos una forma de expresar programas algo más sencilla que este lenguaje y que nos permitirá escribir sobre el papel aquello que después programaremos delante del ordenador: Empezaremos por hacer algoritmos.

Lenguajes de programación

A lo largo de los años, desde la creación de los ordenadores hasta ahora, muchos son los diferentes lenguajes que se han usado para programar. Inicialmente el lenguaje utilizado era el código máquina (un lenguaje que el procesador puede "entender" sin traducción), pero ese "lenguaje" basado en códigos numéricos binarios (sólo con ceros y unos) era difícil de utilizar y los programadores que lo usaban eran casi "seres de otro planeta". Conforme se fueron construyendo más ordenadores hubo que buscar una manera de poderlos programar de una forma más sencilla. Y aparecieron los compiladores, programas capaces de traducir un conjunto de instrucciones escritas en un lenguaje más o menos similiar al humano, al código binario que el procesador es capaz de entender.

En un primer momento los lenguajes de programación eran muy similares al lenguaje máquina, muchas veces no eran más que transcripciones en letras de los los códigos binarios. A ese tipo

de lenguajes se les llama lenguajes de bajo nivel porqué están muy "abajo", muy cerca de la máquina.

Más adelante fueron apareciendo lenguajes más cercanos al ser humano y, por tanto, más alejados de la máquina. Son los lenguajes de alto nivel, los que suelen usarse hoy en día. Pero lenguajes de programación hay muchos, muchísimos y por tanto se clasifican de muy diversas formas. Aunque no entraremos aquí a ver diferentes clasificaciones de lenguajes de programación sí que comentaremos algunos lenguajes de programación para entender un poco mejor qué significa programar.

A continuación hay una lista no exahustiva de lenguajes de programación con una breve explicación sobre ellos.

Fortran Es un lenguaje creado en el año 1957 para facilitar la creación de programas de cálculo matemático.

Algol Creado en 1958, se creó pensando en que fuese independiente del ordenador en que se usase. A pesar de ser bastante completo y de que fue evolucionando, su utilización quedó reducida básicamente al entorno académico. Se considera al Algol como antecesor de los lenguajes C, Pascal, Modula y otros.

Lisp Acrónimo de List Processing es un lenguaje muy diferente a Fortran y Algol. Durante tiempo se consideró como un lenguaje modelo para la investigación en inteligencia artificial.

Cobol Es un lenguaje de programación creado en 1960 por un consorcio formado por fabricantes de ordenadores y el gobierno de los Estados Unidos, con el fin de buscar un lenguaje único que funcionase igual en todo tipo de ordenadores. Es un lenguaje pensado sobre todo para facilitar la creación de programas de gestión y durante mucho tiempo fue el rey en las aplicaciones de las grandes empresas. Todavía hoy pueden encontrarse numerosos programas escritos en cobol en dichas empresas.

Basic (Beginner's All-purpose Symbolic Instruction Code) Es un lenguaje creado a principio de los años 70 con el objetivo de facilitar el aprendizaje de programación. Basado en parte en Fortran, era sencillo de programar, aunque en su momento tuvo muchos críticos por no ser estructurado. Hoy en día una versión de Basic estructurado, el Visual Basic, es habitualmente utilizado por programadores de todo el mundo.

Pascal Creado por el profesor Niklaus Wirth a principio de los años 70, pretendía ser un lenguaje destinado al aprendizaje de lo que se llama programación estructurada. A diferencia del Basic inicial, nada estructurado, los programas en Pascal deben seguir toda una serie de normas destinadas, sobre todo, a facilitar su comprensión y, por tanto, su mantenimiento. Aunque se considera un lenguaje básicamente académico existe una versión orientada a objetos, Delphi, de amplia utilización en entornos empresariales.

C Creado en 1972 por Dennis Ritchie, es famoso por ser el lenguaje de programación con el que se escribió el sistema operativo UNIX. Se considera un lenguaje de nivel medio y hoy en día se utiliza en todas aquellas aplicaciones en la que es más importante que el programa resultante sea rápido que la facilidad de programación. Existe una versión orientada a objetos, el C++.

Ada Fue desarrollado por el Departamento de Defensa de los Estados Unidos para tener un lenguaje común a todos los ordenadores que usaban. Su nombre es en reconocimiento a Augusta Ada Byron, hija del poeta Lord Byron, considerada por algunos como la primera programadora del mundo. Es uno de los primeros lenguajes en permitir la ejecución de varios procesos simultáneamente dentro de un mismo programa. Su sintaxis deriva del Pascal.

Prolog Acrónimo de Programación Lógica, Prolog es un lenguaje de programación que analiza estructuras lógicas y obtiene resultados a partir ellas.

Perl Lenguaje de programación surgido con la idea de facilitar las tareas de administración en UNIX pero usado como lenguaje de propósito general.

Java Creado en el año 1995 por Sun Microsystems, Java es un lenguaje de programación de propósito general pero con unas características que facilitan la creación de programas que funcionen a través de Internet.

JavaScript Nace en el año 1995 cuando Netscape introduce la versión 2.0 de Navigator y lo incluye bajo el nombre de Mocha (posteriormente pasaría a llamarse LiveScript). Finalmente se le bautiza con el nombre de JavaScript (a pesar de no tener relación con Java) para llamar la atención de los medios y la industria de la informática.

Otros lenguajes de programación son PL/1, APL, B, Modula, RPG, Snobol, Eiffel, Simula, Smalltalk, Clipper, PHP, ...

Algoritmos

Como ya decíamos en un párrafo anterior, antes de ponerse a programar es necesario planificar el trabajo a realizar. Nosotros vamos a hacer parte de esa planificación escribiendo en un papel una aproximación del programa que vamos a realizar. A esta aproximación le llamaremos algoritmo. ¿Es necesario escribir un algoritmo antes de escribir el programa? No, de hecho generalmente los programadores suelen escribir directamente sus programas sin el paso previo de la creación del algoritmo. Sin embargo, para aprender a programar es conveniente que nos acostumbremos a escribir primero el algoritmo antes de ponernos con el programa. De esta manera podemos revisar con facilidad lo que hacemos y modificar todo lo necesario antes de programar. Por otra parte, el hecho de escribir el algoritmo primero, nos permite acostumbrarnos a pensar antes de programar y nos dará agilidad para, en un futuro, poder programar directamente.

Por poner un ejemplo imaginemos un viajante que empieza a trabajar en una zona que no conoce. Para poder llegar a todos los clientes que debe visitar un día, deberá primero planificar su desplazamiento sobre el papel, para, de esta manera, no hacer más quilómetros de los necesarios. Cuando lleve un tiempo recorriendo la zona, será capaz de planificar el recorrido en función de los clientes a visitar sin necesidad de usar el papel. De igual manera, para ahorrarnos trabajo y tiempo, nosotros debemos empezar por "planificar" nuestros programas sobre el papel. Una vez tengamos soltura ya podremos hacerlo directamente... ¡Aunque una planificación previa sobre el papel siempre será provechosa!

Resumen

Para programar necesitamos algún sistema que nos permita indicar al ordenador las instrucciones que debe seguir. Para ello utilizamos los lenguajes de programación. Sin embargo, antes de ponernos a programar debemos planificarnos. Una parte de esa planificación será la creación del algoritmo, una aproximación a lo que después será realmente el programa.

Datos

Datos, tipos de datos y variables

Hasta ahora hemos visto programas muy sencillos. Pero es fácil ver que, para hacer programas más útiles, las cosas se complicarán un poco. Continuando con el ejercicio de la receta de la tortilla de patatas, pensemos en una hipotética receta para hacer pasteles de cumpleaños. Como pasteleros, necesitaremos obtener algunos datos para hacer un buen pastel. Necesitaremos, por ejemplo, saber si al homenajeado le gusta o no el chocolate (en caso contrario, haremos un pastel de frutas), el número de comensales y el nombre a poner en el pastel. En función de estos datos, variaremos el procedimiento a seguir.

Así pues, necesitamos almacenar tres datos. Cada uno de ellos se guardará en una variable.

Variables

Para poder trabajar con los datos los almacenaremos en las variables. Una variable viene a ser un almacén donde podemos guardar un dato determinado. De esta manera para cada dato necesitaremos una variable.

Para diferenciar unas variables de otras a cada una le daremos un nombre que será diferente al nombre de cualquier otra variable. A los nombres de las variables también se les llama identificadores, porque la identifican de manera única.

Así podemos tener una variable que se llame precio que guarde un valor (numérico) que será el precio de un determinado artículo (por ejemplo 137). Y una variable que se llame producto que guarde un valor (de texto) que será el nombre del artículo que tiene ese precio (por ejemplo cámara). Es importante diferenciar el nombre de la variable (producto) del valor que almacena (cámara).

En primer lugar guardaremos un Sí o un No en una variable "chocolate". A las variables que sólo pueden almacenar los valores "Sí" y "No" (más bien cierto y falso) se las llama booleanas o lógicas.

George Boole

Nacido el 2 de Noviembre de 1815 en Lincoln, Lincolnshire (Inglaterra), publicó en el año 1854 las leyes del pensamiento sobre las cuales se basan las teorías matemáticas de Lógica y Probabilidad. Boole aproximó la lógica en una nueva dirección, reduciéndola a un álgebra simple, vinculando más lógica y matemáticas. Agudizó la analogía entre los símbolos algebraicos y aquellos que representan formas lógicas. Su álgebra consiste en un método para resolver problemas de lógica que recurre solamente a los valores binarios 1 y 0 y a tres operadores: AND (y), OR (o) y NOT (no). Comenzaba el álgebra de la lógica, llamada Algebra Booleana que ahora encuentra aplicación en la construcción de ordenadores, circuitos eléctricos, programación, etc.

Boole también trabajó en ecuaciones diferenciales, el influyente Tratado en Ecuaciones Diferenciales apareció en 1859, el cálculo de las diferencias finitas, Tratado sobre el Cálculo de las Diferencias Finitas (1860), y métodos generales en probabilidad. Publicó alrededor de 50 escritos y fue uno de los primeros en investigar las propiedades básicas de los números, tales como la propiedad distributiva.

El trabajo de Boole llegó a ser un paso fundamental en la revolución de los ordenadores cuando Claude Shannon, en 1938, demostró como las operaciones booleanas elementales se podían representar mediante circuitos conmutadores eléctricos, y cómo la combinación de estos podía representar operaciones aritméticas y lógicas complejas. Shannon demostró asimismo que el álgebra de Boole se podía utilizar para simplificar circuitos conmutadores.

A continuación, guardaremos el número de comensales en la variable numérica "comensales".

Finalmente, almacenaremos el nombre del homenajeado en una variable de texto, "nombre".

En la realización del programa necesitaremos más variables (el número de huevos, el tiempo de cocción, la temperatura del horno...) pero todos los tipos de datos que veremos a lo largo de este texto se limitarán a los tres que acabamos de ver.

Hemos visto que los datos con los que trabajamos deben guardarse en un espacio al que llamamos variable. En algunos lenguajes de programación para poder utilizar una variable debemos decir antes qué tipos de datos guardaremos dentro. No es el caso de ActionScript, que para diferenciar un texto de un número utiliza sólo las comillas (los textos siempre irán entre comillas). Sin embargo, hay veces en que podemos tener dudas a la hora de decidir si el dato que tenemos es un texto o un número. Vamos a ver algunos ejemplos:

“Hola”	es evidente que es un texto
“Tengo 37 años”	también es un texto
“37”	¿Qué es? ¿Un texto o un número?
37	¿Y esto?

A simple vista podemos decir que los dos treinta y siete son números, pero sin embargo uno lo tenemos puesto entre comillas (como el resto de textos) y el otro no. Las comillas indican que lo que hay entre ellas es un texto, así que el primer 37 es un texto y el segundo un número. En la mayoría de lenguajes de programación esto está muy claro, y si en una variable sólo podemos guardar textos, un “37” siempre será un texto. Y si en una variable sólo podemos guardar números, un 37 siempre será un número. Y, todavía mejor, en una variable de texto no podremos guardar un 37 y en una variable numérica nunca podremos guardar un “37”. Sin embargo en ActionScript esto no es así. No tenemos variables numéricas y variables de texto, sólo tenemos variables que pueden guardar de todo. Y nosotros deberemos preocuparnos de que quede muy claro con qué tipo de valor estamos trabajando si queremos que el resultado de nuestro programa tenga algún sentido.

La falta de tipos no es una característica específica de ActionScript. Pocos lenguajes de Script tienen comprobación de tipos.

Para diferenciarlas, a las variables les daremos un nombre. Los nombres de variables pueden ser cualesquiera que incluyan letras, números y el carácter de subrayado (_), siempre que comiencen por una letra. En general, para evitarnos complicaciones, es recomendable que las variables sean palabras que indiquen que datos se van a almacenar en ellas. Las empezaremos siempre con minúscula y si la palabra debe ser compuesta, la segunda palabra la empezaremos con mayúscula.

Por ejemplo:

edad	Sería una variable numérica donde guardaríamos edades
precio	Sería una variable donde guardaríamos precios
nombreCliente	Sería una variable de texto donde guardaríamos el nombre del cliente

Aunque no siempre es necesario, es conveniente acostumbrarse a indicar, al principio del programa, que variables vamos a usar. En algunos lenguajes de programación este paso es obligatorio, debiendo indicar, también el tipo de la variable. En ActionScript, sin embargo, no lo es. A pesar de ello, es conveniente que nos acostumbremos a hacerlo. El hecho de tener todos los nombres de las variables que usemos definidas al principio del programa nos facilitará el trabajo, especialmente si el programa es muy largo.

Para indicar que variables usaremos en nuestro programa basta con asignarles un valor (valor inicial) al principio del programa. De esta manera tenemos una lista de variables usadas y nos aseguramos de no olvidarnos de poner un primer valor en la variable. Sin embargo, en Actionscript tenemos una manera para definir nuevas variables que es mediante la sentencia var. Veremos como usarla cuando hablemos de funciones.

Nombres de variables

Aunque es recomendable seguir las pautas que se han indicado en el texto, apenas hay límites a la hora de poner nombres. Así, las únicas normas “reales” son:

- La longitud está limitada a 256 caracteres.
- Siempre debe empezar por una letra.
- Sólo puede contener letras, números y el carácter de subrayado
- No admite espacios, letras acentuadas, eñe, ni ce cedilla (ç)

Así pues serían nombres de variables correctos:

Lksjdflskfsdlknslsdf

ASkjoIWEIjcn2424klj32

r344231233

En_un_lugar_de_la_Mancha

Sin embargo, si tuviésemos que hacer un programa donde alguna de estas variables se usase varias veces, el tenerlas que escribir en cada ocasión alargaría enormemente el tiempo necesario para escribir el programa.

Por otra parte también sería correcto tener nombres de variables como:

A1

A2

A3

…

Pero en este caso resulta casi imposible saber para qué estamos usando cada una de las variables. Y es muy importante no equivocarse a la hora de usarlas, pues si por error mezclamos edades con quilos el resultado que obtengamos no será el que deseamos.

Variables booleanas

Aunque ya se ha comentado no está de más insistir en el tipo de datos booleano, porqué se usará mucho más a menudo de lo que pueda parecer. Una variable booleana sólo puede guardar dos valores: verdadero o falso (true - false). Tal vez pueda parecer que un tipo de datos que sólo tiene dos valores ha de ser poco útil pero no es así. Muy a menudo en nuestros programas nos veremos obligados a hacernos preguntas cuya respuesta será sí o no. Por ejemplo, si al homenajeado le gusta chocolate. Y si la respuesta es sí, guardaremos verdadero en una variable booleana llamada chocolate. O al hacer el cálculo del precio de un producto, podemos tener una variable booleana donde guardar si ese producto está en oferta o no. Y si en oferta tenemos un cierto, calcularemos el precio después de aplicar el descuento correspondiente y sino calcularemos el precio sin descuento.

En posteriores apartados de este mismo material veremos como el tipo de datos booleano es muy usado en programación. Y deberemos acostumbrarnos a operar datos de este tipo.

Asignación, lectura y escritura de datos

Hemos visto que podemos guardar datos en las variables. ¿Cómo se hace eso? Teniendo en cuenta que nuestros programas los haremos con ActionScript, tendremos dos maneras de guardar los datos en las variables. Por una parte, cuando en el programa nos interesa guardar un valor en una variable podemos utilizar la asignación.

En general la asignación será:

<variable> = <valor>

Donde <variable> será un nombre que defina a esa variable de manera única (sin posibilidad de confusión con otra). Y <valor> será un valor cualquiera.

Esta expresión se leería de la siguiente manera:

<variable> toma por valor <valor>

o

Asignamos el valor <valor> a la variable <variable>

o

Guardamos el valor <valor> en la variable <variable>

Hay que tener presente que en este caso el signo = no representa una igualdad, por eso decimos: toma por valor, que de alguna manera pretende expresar que el valor se guarda en la variable.

Con esta información ya podemos guardar algunos datos en nuestras variables:

```
edad = 37
precio = 125.37
nombreCliente = "César Muñoz Marín"
```

Por otra parte podremos permitir que el usuario (aquella persona que utiliza nuestro programa) introduzca directamente los datos en las variables. Cuando hagamos eso diremos que leemos los datos del teclado (el usuario usará el teclado para escribir) y en nuestros algoritmos escribiremos leer.

Así, cuando queramos obtener un dato del usuario escribiremos:

leer(variable)

Y la variable será cualquiera, una usada anteriormente o una que usemos por primera vez.

También nos interesará que el usuario vea en pantalla el resultado de ejecutar el programa. Para hacer esto en nuestros algoritmos utilizaremos la instrucción escribir:

escribir (<expresión>)

Donde <expresión> será un valor o una variable

De esta manera el resultado de una operación podrá presentarse en la pantalla para que el usuario pueda verlo. En el ejemplo que viene a continuación el programa sería:

```
{
    valor = 0;
    resultado = 0;
    leer(valor);
    resultado = valor * 2;
    escribir(resultado);
}
```

Ver y probar ejemplo 1

Operadores y operaciones

Ahora que hemos visto como guardar datos en las variables, vamos a ver cómo operar con esos datos. Empezaremos con las operaciones matemáticas básicas:

Operaciones matemáticas

Suma	<valor> + <valor>
Resta	<valor> - <valor>
Multiplicación	<valor> * <valor>
División	<valor> / <valor>
Módulo (resto de la división)	<valor> % <valor>

La operación %

El módulo (o resto de la división) es una operación que dados dos números nos dice cual es el resto de la división entera entre ambos operandos.

Así, la operación

```
c = 4 % 2
```

Almacenará el valor 0 (el resto de dividir 4 entre 2 es 0) en la variable c.

Y la operación

```
c = 15 % 6
```

Almacenará el valor 3 en la variable c (15 dividido por 6 es 2 y el resto 3).

Así podríamos tener:

```
4 + 2
4 - 2
4 * 2
4 / 2
4 % 2
```

Y si usamos variables nos podríamos quedar con:

```
a = 4        // La variable a toma por valor 4.
b = 2        // La variable b toma por valor 2.
c = a + b    // La variable c toma por valor el resultado de sumar el
             // contenido de la variable a al contenido de la variable b.
```

Después de realizar la última asignación en la variable c tendremos almacenado el valor 6.

Y si ahora hacemos:

```
c = a * b
```

En la variable c tendremos almacenado el valor 8.

Autoincremento y autodecremento

Hay dos operaciones que son un poco especiales. Son el autoincremento y autodecremento. Vamos a verlas:

Autoincremento

Se representa de la siguiente manera:

<variable>++

Por ejemplo:

i++

¿Qué significa? De hecho i++ equivale a escribir i = i + 1. Si miramos esta expresión con una "mirada matemática" puede parecer una expresión imposible

(i es igual i + 1 o que sería similar: 5 es igual a 5 + 1 => 5 es igual a 6).

Sin embargo no debemos olvidar que el símbolo = en este caso no representa igualdad sino asignación y que, en realidad, esta expresión deberíamos leerla como guardamos en `i` el valor resultante de sumar 1 al valor actual de `i` (o `i` toma por valor `i` + 1).

Así, si antes de la operación i++ la variable i valía 5, después de esta operación su valor será 6.

```
{
i = 33;
i++;
escribir (i);
}
```

¿Qué se escribiría en pantalla después de ejecutar este algoritmo? La respuesta es sencilla, 34.

Autodecremento

Es una operación similar a la de autoincremento pero con la diferencia de que en vez de sumar restamos. Se representaría de la siguiente manera:

<variable>--

Por ejemplo:

i--;

Así pues, si el valor de i antes de ejecutar esta operación era 7, después de ejecutarla será 6.

Operaciones con cadenas

ActionScript nos permite hacer algunas operaciones con textos (cadenas de caracteres). Nosotros aquí sólo vamos a ver una de ellas:

Concatenación

<cadena> + <cadena>

El resultado es una cadena de caracteres que contiene las dos que hemos concatenado. Ejemplo:

c = "hola" + "adiós"

Después de realizar esta operación en la variable c se habrá almacenado el valor "holaadiós" (sin espacio, puesto que ninguno de los dos valores lo tenía).

Comparaciones

En el caso de la receta del pastel teníamos varias comparaciones. De alguna manera deberíamos comprobar si al homenajeado le gusta el chocolate o no. En un programa la secuencia sería aproximadamente esta:

Preguntamos al homenajeado si le gusta o no el chocolate y guardamos su respuesta en una variable.

Si el valor guardado en esa variable es No (comparamos el valor de la variable con la cadena “No”) haremos el pastel de frutas.

Otro caso guardaría relación con el número de comensales. Supongamos que podemos hacer el pastel para 6 o para 10 personas. Primero preguntaremos cuantos comensales habrá y guardaremos la respuesta en una variable. Después miraremos si el valor de esa variable es mayor que 6 y entonces tendremos que usar los ingredientes necesarios para hacer el pastel de 10 personas.

En muchas ocasiones nos interesa comparar dos valores. Sea para saber cuál de los dos es el mayor, como para saber si son iguales o no, las operaciones de comparación son de las más utilizadas en los programas informáticos. Las operaciones de comparación que podemos usar son:

Igualdad	<valor> == <valor>
Mayor que	<valor> > <valor>
Menor que	<valor> < <valor>
Diferente	<valor> != <valor>
Mayor o igual que	<valor> >= <valor>
Menor o igual que	<valor> <= <valor>

La comparación de igualdad se indica con dos signos = seguidos (==). De esta manera se diferencia de la asignación. Hay que tener siempre presente esta diferencia para evitar errores indeseados.

Es muy importante no confundir la asignación, que implica guardar en, con la igualdad, que implica comparación.

La comparación como operación

Puede resultar difícil de entender que el resultado de una comparación se guarde en una variable. Pero deberíamos ver la comparación como una operación entre dos valores que da como resultado un valor booleano: cierto o falso. Si partimos de la comparación como operación podemos decir que si tenemos la expresión:

c = 5 == 4

Como el resultado de operar con la igualdad 5 con 4 es falso, el valor que se almacenará en la variable booleana c es falso.

O visto de otra manera:

Es falso que 5 == 4, por tanto si pongo c = 5 == 4 el valor que se almacenará en c es falso.

En cambio en la expresión:

c = 5 > 4

El valor que se almacenará en la variable c será cierto pues 5 es mayor que 4.

El resultado de hacer una comparación siempre será un valor booleano.

Prioridad de las operaciones

Cuando tenemos que realizar varias operaciones (en una misma línea) hemos de tener presente en qué orden se realizarán esas operaciones, porque dependiendo del orden, el resultado final será diferente. Veamos un ejemplo:

c = 4 + 4 / 4

En este caso si primero hacemos la suma (4 + 4 --> 8) y después la división (8 / 4 --> 2) el valor que se almacenará en la variable c será 2.

Sin embargo, si primero hacemos la división (4 / 4 --> 1) y después la suma (4 + 1 --> 5) el valor que se almacenará en la variable c será 5.

¿Cuál de los dos resultados es el correcto?

Si hubiésemos usado paréntesis la cosa estaría muy clara:

Si tenemos la expresión c = (4 + 4) / 4

El valor que se guardará en la variable c será 2

Sin embargo si tenemos la expresión c = 4 + (4 / 4)

El valor que se guardará en la variable c será 5

Pero si no hemos usado paréntesis necesitamos saber en que orden realiza las operaciones el ordenador para poder saber cual será el resultado de la operación. Ese orden es el mismo que utilizan las calculadoras científicas.

La siguiente lista muestra los operadores de más a menos prioridad:

* / % (mayor prioridad)

+ - (menor prioridad)

La multiplicación, la división y el resto tienen la misma prioridad y la suma y la resta también. En caso de que se encuentren dos operadores de la misma prioridad en una misma expresión se evalúan de izquierda a derecha (primero se calcula la operación que hay más a la izquierda en la expresión).

Así, si tenemos

c = 4 * 4 / 2;

Primero se calculará 4 * 4 y después se dividirá el resultado por 2.

Los operadores de comparación se evalúan en último lugar. En caso de que hayan dos operadores de la misma prioridad juntos, se evaluará primero el que esté más a la izquierda en la expresión.

Si tenemos en cuenta este orden resulta fácil saber que el valor que se almacenará en la variable c de la expresión c = 4 + 4 / 4, será 5.

Operadores lógicos o booleanos

Igual que tenemos un tipo de datos booleano tenemos unos operadores booleanos que nos permiten operar con los datos de ese tipo. Las operaciones básicas son:

Y lógico

<valor> **&&** <valor>

Da resultado cierto sólo si los dos valores comparados son ciertos.

Así tenemos cierto **&&** cierto **=>** cierto, pero en el resto de casos el resultado siempre es falso.

a	b	a && b
Cierto	Cierto	Cierto
Cierto	Falso	Falso
Falso	Cierto	Falso
Falso	Falso	Falso

O lógico

<valor> **||** <valor>

Da resultado cierto si cualquiera de los dos valores comparados es cierto.

Así si tenemos cierto **||** cierto **=>** cierto, sólo en el caso en los dos valores sean falso el resultado es falso.

a	b	a || b
Cierto	Cierto	Cierto
Cierto	Falso	Cierto
Falso	Cierto	Cierto
Falso	Falso	Falso

Leyes de la lógica

Cuando se trabaja con expresiones booleanas (como por ejemplo (x == y o z == r)) conviene tener presente las leyes de la lógica porque muchas veces nos permitirán simplificar las expresiones o, simplemente, escribirlas de forma más clara.

Son estas: (el símbolo <=> se puede leer como *equivale*, es decir, que la expresión de la derecha es equivalente a la expresión de la izquierda)

Dados unos predicados p, q y r

Ley de la doble negación

no (no p) <=> p

Leyes conmutativas

p y q <=> q y p

p o q <=> q o p

En las expresiones booleanas se cumple la propiedad conmutativa, de manera que el orden de los operandos no altera el resultado.

Leyes asociativas

p y (q y r) <=> (p y q) y r

p o (q o r) <=> (p o q) o r

También se cumple la propiedad asociativa: no importa de que manera se agrupen los operandos, siempre y cuando el operador sea el mismo.

Leyes distributivas

p y (q o r) <=> (p y q) o (p y r)

p o (q y r) <=> (p o q) y (p o r)

Otra propiedad que se cumple es la distributiva.

Leyes de dominación

p y falso <=> falso

p o cierto <=> cierto

En la y, si cualquiera de los dos operandos es falso el resultado es falso. Por tanto si uno de los operandos es falso el resultado será falso independientemente del valor del otro operando.

Una cosa similar pasa en el caso de la o.

Leyes de elemento neutro

p y cierto <=> p

p o falso <=> p

Siguiendo el razonamiento de las leyes anteriores, cuando en una operación y un operador vale cierto, si p es cierto el resultado será cierto y si p es falso el resultado será falso.

Leyes de elemento inverso

p y no p <=> falso

p o no p <=> cierto

En la y, si cualquiera de los dos operandos es falso el resultado es falso. Por tanto si p es cierto, no p será falso y por tanto el resultado será falso. Pero si p es falso el resultado también será falso.
Una cosa similar pasa en el caso de la o. En este caso si uno de los dos es cierto el resultados será cierto.

Leyes de De Morgan

no (p y q) <=> no p o no q

no (p o q) <=> no p y no q

Las leyes de De Morgan nos dicen que (al contrario de lo que creeríamos) la negación de dos predicados no es equivalente a los dos predicados negados. Es decir que esta afirmación:

no (p y q) <=> no p y no q **No es correcta**

Para finalizar es conveniente tener presente que en caso que en una expresión tengamos una operación **y** y una **o** primero se evaluará la **y** y después la **o** (la **y** tiene más prioridad).

Ejemplo:

¿Podemos simplificar la siguiente expresión?

p y (q o no p)

Sí. Podemos aplicar la propiedad distributiva y nos quedaremos con:

(p y q) o (p y no p)

Pero sabemos que p y no p es falso (por la ley de elemento inverso) así que nos quedaría

(p y q) o falso

Y aplicando la ley de elemento neutro nos queda:

p y q

No lógico

! **<valor>**

Convierte cierto en falso y falso en cierto.

a	!a
Cierto	Falso
Falso	Cierto

También los operadores de comparación tienen diferentes prioridades. En concreto se ejecutará primero la *negación*, después el *y* y finalmente el *o*. Así el orden de prioridades es:

!
&&
||

Resumen y actividades

Resumen

Los datos son la razón de ser de todos los programas. Debemos tener claro cómo los trataremos y qué podemos hacer con ellos. En estas páginas hemos visto algunos tipos de datos y algunas de las operaciones que podemos realizar con ellas. También hemos visto dónde guardaremos los datos mientras los estemos tratando en nuestro programa: las variables. Cada variable tendrá un nombre que la identifique.

Los datos con los que trabajaremos son de diferentes tipos; hemos visto tres, los numéricos, los de texto y los booleanos (un tipo de datos "especial" que sólo admite los valores cierto y falso).

Algunas palabras que podríamos definir en esta etapa son:

Variable	Lugar donde almacenaremos los datos que usaremos a lo largo de nuestro programa.
Operador	Símbolo que realiza una tarea, u operación, sobre uno o varios datos, denominados operandos.
Ejecutar (un programa)	Ponerlo en funcionamiento.
Expresión	Conjunto de operadores y operaciones.
Usuario	La persona que utilizará nuestro programa.

Actividades

1.- ¿Cuáles de los siguientes nombres de variables son correctos y cuales no?

12345

abcd

enUnLugar

en1Lugar

_variable

Variable_1

NumMaños

Variable-1

2.- ¿Qué valor se guardará en la variable a después de ejecutar el siguiente algoritmo?

```
{
      x = 5;
      y = 7;
      z = 3;
      a = x + y *z;
}
```

3.- ¿Qué hace el siguiente algoritmo?

```
{
      leer (a);
      leer (b);
      c = a + b;
      escribir (c);
}
```

4.- Teniendo en cuenta el algoritmo que acabas de ver, escribe un algoritmo que lea dos valores y escriba el resultado de multiplicar uno por otro.

5.- ¿Qué crees que escribiría el siguiente algoritmo?

```
{
      a = 5;
      b = “hola”;
      c = a + b;
      escribir (c);
}
```

¿Por qué?

6.- ¿Qué valor se almacenaría en la variable c en cada caso?

c = cierto || falso && cierto

c = falso || cierto && cierto

c = !cierto && cierto || falso

c = 5 < 8

c = 6 + 2 == 11 - 3

Estructuras de control sencillas

Introducción

Hemos visto que los programas van ejecutando las instrucciones que nosotros escribimos. Hasta ahora, en los diferentes ejemplos que hemos ido viendo, la ejecución se hacía instrucción a instrucción, la primera se ejecuta primero, la segunda a continuación, etc. Aunque sin duda es un sistema sencillo y fácil de entender no permite realizar grandes y complejos programas.

Al orden en que se ejecutan las diferentes instrucciones le llamamos flujo y a las estructuras que permiten variar ese flujo les llamamos estructuras de control de flujo o, simplificando, estructuras de control.

Así pues para escribir los programas no sólo deberemos tener en cuenta las operaciones e instrucciones que necesitemos, sino que también que estructuras de control de flujo usaremos. En esta etapa veremos las estructuras de control de flujo más sencillas, la secuencial, la condicional sencilla (**si**) y la condicional doble (**si – sino**).

Estructura secuencial

Se podría decir que la estructura secuencial es la falta de estructura, ya básicamente consiste en escribir las instrucciones una detrás de otra, de manera que se ejecutarán en el mismo orden en que las hayamos escrito.

Supongamos que queremos calcular la suma y la resta de dos valores (5 y 15). El programa que haríamos podría ser como este:

```
{
    a = 5;
    b = 15;
    suma = a + b;
    resta = a - b;
}
```

Este es un caso típico de estructura secuencial. Las instrucciones se ejecutan una detrás de otra hasta obtener el resultado final.

Hay un par de cosas que llaman la atención en este pequeño algoritmo que acabamos de ver. En un primer lugar después de cada instrucción hay un punto y coma (;). En muchos lenguajes de programación se utiliza este símbolo para indicar el final de la instrucción y separar una instrucción de la siguiente. Su utilización es obligatoria y por tanto no debemos olvidar nunca el punto y coma al acabar una instrucción.

Por otra parte, al empezar el algoritmo hay un abrir llave y un cerrar llave al acabar. En general, se utilizan las llaves para indicar dónde empieza y dónde acaba una parte de código que debe ir junta. En las siguientes estructuras de control veremos más claramente la utilidad de estos símbolos.

Observa y prueba el siguiente ejemplo de estructura secuencial.

Para probarlo debes introducir dos números. Una vez lo hayas hecho aprieta el botón *Iniciar* y verás que la primera llave cambia de color (pasa de negro a rojo). A partir de ese momento cada vez que aprietes el botón *Avanzar* se ejecutará una instrucción. En el cuadrado donde pone *Valor de las variables* podrás ver el valor que van tomando las diferentes variables conforme va avanzando el programa.

Ver y probar ejemplo 2

Estructura condicional sencilla

A veces interesa que no siempre se ejecute alguna parte del programa. Por ejemplo, cuando queríamos hacer un pastel de cumpleaños, uno de los datos que nos daban era si al homenajeado le gustaba o no el chocolate. Decíamos que si no le gustaba el chocolate haríamos el pastel de frutas. Así, dependiendo del valor que haya en una variable (cierto o falso según le guste o no el chocolate) deberemos realizar unas acciones (para hacer un pastel de chocolate) u otras (que nos permitan hacer un pastel de frutas). De alguna manera esto es lo que nos permiten hacer las estructuras condicionales.

Vamos a ver otro ejemplo. Si en una variable se almacena la edad de una persona y queremos mostrar el nombre de una película sólo en el caso de que esa persona sea menor de 15 años necesitaremos comprobar si la edad es correcta antes de mostrar el título.

Esta estructura condicional se escribe de la siguiente manera:

```
si (condición) {
      instrucciones
}
```

Donde la condición estará generalmente formada por una expresión booleana.

Siguiendo con el ejemplo anterior y suponiendo que la edad de la persona que ejecuta el programa se almacena en una variable llamada *edad* las instrucciones se escribirían de la siguiente manera:

```
{
      leer (edad);
      si (edad < 15) {
            escribir ("Alibabá y los 40 ladrones");
      }
      escribir ("Adiós");
}
```

En este caso, se leerá la edad del usuario y si tiene menos de 15 años se escribirá el nombre de la película. Tendremos dos posibles casos:

Edad introducida	***En pantalla se escribirá***
10	`Alibabá y los 40 ladrones` `Adiós`
20	`Adiós`

¿Cómo se sabe que la instrucción Escribir ("Alibabá y los 40 ladrones") debe ejecutarse sólo cuando se cumple la condición?

¿Porqué la condición no afecta también a la instrucción Escribir ("Adiós")?

La respuesta es simple. Después de la condición tenemos un abrir llave. Si la condición se cumple se ejecutará aquello que hay entre las llaves y si no se cumple no se ejecutará.

Las llaves

Es importante tener claro el funcionamiento de las llaves. En la estructura de control Si (recordemos:

Si (condición) {

Instrucciones

}

) las instrucciones tienen que ir entre llaves. Sólo de esa manera el ordenador es capaz de saber que instrucciones debe ejecutar cuando se cumple la condición.

En el siguiente ejemplo se puede ver el funcionamiento de una estructura condicional simple. Si el valor que introducimos es menor que 15 el flujo del programa nos llevará a la instrucción que escribe “Alibabá y los 40 ladrones”. Sin embargo si el número introducido es mayor o igual a 15, el programa saltará esa instrucción y esa frase no se escribirá en la pantalla.

Prueba el ejemplo introduciendo diferentes valores y comprobando que funciona tal y como se espera.

Igual que en el ejemplo anterior hay que introducir el número que se desee y a continuación apretar **Iniciar**. Una vez iniciado el programa (se sabe, porqué la instrucción que se está ejecutando en ese momento aparece en color rojo), se puede ir avanzando apretando el botón sobre **Avanzar**.

Ver y probar ejemplo 3

Estructura condicional doble

Siguiendo con el ejemplo anterior, podríamos querer que si no se cumple la condición se nos informase de ello, por ejemplo con un mensaje. En este caso utilizaríamos la estructura condicional doble cuya sintaxis es la siguiente:

```
Si (condición) {
    Instrucciones a realizar si se cumple la condición;
} sino {
    Instrucciones a realizar si no se cumple la condición;
}
```

Por ejemplo, siguiendo con el algoritmo que ya habíamos hecho.

```
{
    leer (edad)
        si (edad < 15 ) {
            escribir (“Alibabá y los 40 ladrones”);
        } sino {
            escribir (“No cumple con la edad”);
        }
    escribir (“Adiós);
}
```

Aquí las posibilidades serían:

Edad introducida ***En pantalla se escribirá***

10	`Alibabá y los 40 ladrones` `Adiós`
20	`No cumple con el criterio de edad` `Adiós`

A diferencia del **si** (donde si la condición no se cumplía no se hacía nada) en el **si** – **sino** si la condición no se cumple se ejecutan las instrucciones que hay en el **sino**.

Resumen

Las estructuras de control nos permiten hacer programas que ejecuten diferentes instrucciones según los datos que tenemos. De esta manera podemos adecuar los programas a diferentes casos.

Actividades

1. Escribe un algoritmo que lea 2 valores (ancho y alto de un rectángulo) y escriba el área de ese rectángulo.
2. Escribe un algoritmo que lea 1 valor (el diámetro de un círculo) y escriba la longitud de la circunferencia.
3. Escribe un algoritmo que calcule el valor absoluto de un número (el valor absoluto de un número es ese número con signo positivo, es decir que si el número es mayor que cero no hay que hacer nada y si es menor que cero hay que multiplicarlo por -1).
4. Escribe un algoritmo que lea dos números y escriba el mayor de los dos.
5. Escribe un algoritmo que lea tres números y escriba el mayor de los tres.
6. Escribe un algoritmo que lea tres números y escriba los tres de mayor a menor.

Estructuras de control complejas (bucles)

Introducción

Una de las mayores utilidades de los ordenadores es su capacidad de realizar tareas repetitivas en muy poco tiempo y con mucha facilidad. Hasta ahora las estructuras de control que hemos visto, no permitían la repetición sino que tenían un principio y un fin claro que se veía incluso sobre el papel.

En esta etapa vamos a ver estructuras que nos van a permitir repetir cálculos un numero determinado de veces. Las utilizaremos a menudo a pesar de que son más complicadas y “peligrosas” que las anteriores.

Para

Hay casos en que nos interesa repetir una operación (o un conjunto de operaciones) un número determinado de veces. Por ejemplo, si queremos calcular la media de 20 números nos iría muy bien poder pedir los 20 números de una vez sin tener que escribir una a una las 20 instrucciones de leer. Además podríamos incluso ahorrarnos variables. Veamos como haríamos este algoritmo a partir de lo que hemos aprendido hasta ahora.

Nota: Para simplificar se hacen los cálculos con sólo 5 valores. Es fácil imaginar como sería para 20 valores.

```
{
    leer (valor1);
    leer (valor2);
    leer (valor3);
    leer (valor4);
    leer (valor5);
    suma = valor1 + valor2 + valor3 + valor4 + valor5;
    media = suma / 5;
    escribir (media);
}
```

Con un **para** esto se haría de la siguiente forma (ahora para 20 números):

```
{
    suma = 0;
    para (i = 1; i <= 20; i++) {
        leer (valor);
         suma = suma + valor;
    }
    media = suma / 20;
    escribir (media);
}
```

suma = suma + valor

Cuando vimos la operación de autoincremento vimos el significado de i = i + 1; Decíamos entonces que lo que debíamos leer es que la variable i guardará el valor actual de i más 1.

Éste es un caso similar. En la variable suma guardaremos el resultado de sumar los valores que haya en las variables suma y valor.

Así si tenemos:

```
{
    suma = 3;
    valor = 10;
    suma = suma + valor;
    escribir (suma);
}
```

En la pantalla el valor que se escribiría sería 13.

La sintaxis del **para** sería la siguiente:

```
para ( variable = valor_inicial; variable <= valor_final;
variable = variable + incremento) {
      instrucciones;
}
```

El funcionamiento del **para** hace que una vez se llega a la llave de finalización se vuelve al **para**, se comprueba que no hemos acabado y se vuelven a ejecutar todas las instrucciones.

Donde:

variable: Es la variable que se irá incrementando a cada vuelta del bucle.

valor_inicial: Valor que se le asigna a la variable al iniciarse el bucle.

valor_final: Valor máximo al que puede llegar la variable.

incremento: Valor en que se incrementará la variable a cada vuelta.

En este ejemplo se pueden ver los diferentes resultados obtenidos al cambiar los diferentes valores, el inicial, el final y el incremento.

Ver y probar ejemplo 4

Mientras

La instrucción **para** nos permite repetir un conjunto de instrucciones un número determinado de veces.

Sin embargo hay ocasiones en que nos interesa poder repetir esas instrucciones hasta que se produce un hecho determinado. Por ejemplo si no sabemos con exactitud cuantos valores vamos a introducir para calcular la media podría interesarnos acabar el cálculo al entrar un valor negativo. Hay además, cálculos que por sus características no pueden hacerse con un **para**. En un juego, un marcianito deberá moverse (para lo cual programaremos un conjunto de instrucciones) hasta que reciba un disparo. El conjunto de instrucciones que permiten moverse al marcianito deberán repetirse, no un número determinado de veces, si no hasta que reciba el disparo. Hay otros ejemplos en los que si bien el **para** podría usarse no es la elección más adecuada. Supongamos que hacemos un programa que dado un número nos dice si es primo o no. ¿Cómo lo haríamos? Es fácil, sólo tenemos que comprobar si ese número es divisible por alguno de los que son más pequeños que él. Así, si tenemos el 15 probaremos si es divisible por 2 (no), por 3 (sí), por el 4 (no), por el 5 (sí), por el 6 (no), … así hasta llegar al 14. Se puede ver con facilidad que cuando llegamos al 3, como ya sabemos que es divisible, ya podríamos

acabar. Con el **para** obligatoriamente deberíamos hacer todos los números. Con el **mientras** veremos que podemos acabar antes.

La sintaxis del **mientras** es la siguiente:

```
mientras (condición) {
    instrucciones;
}
```

Como se puede apreciar, esta instrucción tiene una sintaxis más sencilla que la del **para**. Esto hace que sea más potente, pero a la vez, algo más compleja de utilizar. Para entender el **mientras** podríamos leer que mientras se cumpla la condición (mientras la condición sea cierta) se repetirán las instrucciones.

Antes de entrar a ver ningún ejemplo hay un detalle que debemos tener siempre presente cuando usemos un **mientras**. Las instrucciones que hay entre las llaves deben hacer alguna operación que nos acerque, cada vez más, al final del bucle. Un "antiejemplo":

```
{
    valor = 1;
    mientras ( valor <= 4 ) {
            escribir ( "Hola");
    }
}
```

En este caso el bucle nunca se acabaría porque *valor* nunca llegaría a valer 4 (*valor* siempre sería menor o igual que 4). Por tanto nuestro programa se quedaría "colgado". Veamos ahora un ejemplo correcto:

```
{
    valor = 1;
    mientras (valor <= 4) {
            escribir ("Hola");
            valor++;
    }
}
```

En este caso, cada vez que se ejecutan las instrucciones que hay dentro del bucle se incrementa la variable *valor* y, por tanto, cada vez estamos más cerca de que el bucle se acabe.

Vamos a ver ahora un ejemplo más elaborado, vamos a calcular si un número es primo o no.

Enfoquemos primero el problema:

Datos de los que disponemos: Un número entero entre 1 y 999999999

```
leer (numero);
```

¿Qué debemos hacer?: Comprobar si ese número es divisible por alguno de los que son más pequeños que él. Si al hacer la división, el resto es 0, eso quiere decir que el número es divisible y, por tanto, no es primo.

```
si (numero % menor == 0) {
     esprimo = falso;
}
```

Faltan más cosas. La variable menor debe tener un valor. Inicialmente será 2. Además supondremos que el número es primo mientras no podamos demostrar lo contrario:

```
leer (numero);
menor = 2;
esprimo = cierto;
```

¿Cuándo se acabará el **mientras**? El **mientras** se acabará cuando *esprimo* valga *falso* o cuando *menor* sea igual a *numero* (en ese caso se habrán acabado todos los números). ¿Cómo expresaríamos esa condición?

```
(esprimo == falso o menor == numero)
```

¿Esta sería la condición que deberíamos poner en el **mientras**?. No. Debemos recordar que el **mientras** se repite mientras la condición sea cierta. Y sin embargo hemos escrito la condición que debe cumplirse para saber que hemos acabado. Curiosamente es justo lo contrario de lo que necesitamos. Pero usando las leyes de la lógica pasar de una expresión a otra no es difícil. Así pues la condición que deberemos poner en el **mientras** será:

```
mientras (esprimo != falso y menor != numero)
```

Si intentamos leer la expresión veremos que tiene más sentido de lo que parece: "mientras *esprimo* sea diferente de *falso* (o sea, *verdadero*) y *menor* diferente de *numero* volver a hacer los cálculos" o, dicho de otra manera, "mientras no encontremos un divisor y no hayamos probado todos los números probamos otra vez".

Vamos a escribir ahora el algoritmo según lo que hemos visto:

```
{
  leer (numero);
  menor = 2;
    esprimo = cierto;
    mientras (esprimo != falso y menor != numero) {
      si (numero % menor == 0) {
        esprimo = falso;
      }
    }
    si (esprimo == cierto) {
      escribir ("El número introducido SI es primo");
    } sino {
      escribir ("El número introducido NO es primo");
    }
}
```

¿Está ya completo? No, hay una cosa que falta… Nunca vamos hacia el final del bucle. Para poder acercarnos cada vez más a la finalización del bucle, necesitamos incrementar el valor de la variable *menor* para que probemos todos los números menores que el que intentamos saber si es primo. En el ejemplo en flash está ya bien resuelto, sin embargo, antes de mirarlo vale la pena intentar encontrar la solución.

Ver y probar ejemplo 5

Resumen

Los bucles nos permiten repetir instrucciones sin necesidad de volverlas a escribir. Sin embargo hemos de tener presente que las instrucciones que hay dentro del bucle siempre tienen que llevarnos hacia la conclusión del mismo.

Aunque no es evidente, el **para** es una versión simplificada del **mientras**. Un buen ejercicio para comprobar si se ha comprendido correctamente el funcionamiento de ambas estructuras, es escribir un programa usando el **para** y, a continuación, repetirlo usando el **mientras**.

Actividades

1. Usando el **para** escribir un algoritmo que escriba 50 veces en pantalla la palabra “Flash”
2. Escribir un algoritmo que usando el **para** nos permita calcular el factorial de un número dado. Hay que tener en cuenta la fórmula del factorial:

   ```
   n! = 2 * 3 * 4 * … * (n-1) * n;
   ```

3. Usando el **mientras** escribir un algoritmo que calcule todos los divisores de un número dado.
4. ¿Se podría haber hecho el programa anterior usando un **para** en vez de un **mientras**?
5. Escribir un algoritmo que escriba los 100 primeros números de la serie de Fibbonaci. Dicha serie está definida como:

   ```
   a0 = 1;
   a1 = 1;
   a2 = a0 + a1;
   a3 = a1 + a2;
   …
   an = an-1 + an-2;
   ```

Introducción al ActionScript

Introducción

El ActionScript es el lenguaje de programación de Flash. Permite que un programa de animación se convierta en una potente herramienta de creación de aplicaciones interactivas para la web. La programación facilita la realización de ciertas tareas que de otra manera serían impensables. Sólo el hecho de poder hacer cálculos matemáticos ya es una utilidad de la programación pero, sin embargo, el que los objetos de la animación atiendan a eventos del ratón, o el poder hacer que dos objetos que se mueven libremente por la pantalla sepan cuando chocan entre ellos o, simplemente conseguir que un determinado objeto siga al ratón por la pantalla, son utilidades que esconden tras de sí, una sencilla (o a veces complicada) programación en ActionScript.

No es el objetivo de este material conseguir grandes programadores en ActionScript. El objetivo principal es aprender a programar. Pero como para aprender a programar es necesario practicar vamos conocer ActionScript para poderlo utilizar en nuestro aprendizaje

Tipos de datos y operaciones

Podemos decir que en ActionScript sólo hay dos tipos de datos, los numéricos y los de texto. En los numéricos podemos englobar todos los números, tanto enteros como con decimales (en este último caso deberemos tener presente que el separador decimal es el punto en vez de la coma). Por otra parte para diferenciar datos numéricos de datos de texto usaremos las comillas en este último caso. Así serán datos numéricos:

```
1
21
321.55
0.012
```

Y serán datos de tipo texto:

```
"hola"
"chechar"
"25"
```

A diferencia de otros lenguajes donde al crear una variable es obligatorio especificar que tipo de datos se almacenará en ella, en ActionScript esto no es necesario y una misma variable puede guardar datos numéricos y de texto en cualquier momento. Esto a la larga nos puede crear un pequeño problema cuando trabajamos con números y es que podemos encontrarnos con que Flash interpreta números como si fuesen texto. Por ejemplo, supongamos que tenemos en una variable (llamémosla *variable1*) guardado el valor 1 y en otra (*variable2*) el valor "1". A pesar de que uno de los valores es texto y el otro numérico y en principio no podríamos operar con ellos (como quien suma peras y manzanas) resulta que si escribimos

```
Variable3 = variable1 + variable2;
```

a Flash le parecerá una operación correcta y la realizará. ¿Cómo? ¿Qué operación hará? Pues convertirá el número en un texto y realizará la operación + entre texto, es decir, la concatenación.

Por tanto en la variable *Variable3* se almacenará el texto "11".

En cuanto a las operaciones que podremos realizar con los datos en ActionScript, decir que son muchas pero que, para nuestros propósitos, nos servirá saber que sirven todas las que vimos en el apartado operadores y operaciones.

Para finalizar un apunte importante. En ActionScript no se leen ni se escriben los valores. Generalmente lo que hasta ahora llamábamos leer ahora será asignar a una variable el valor almacenado en un campo de texto (que tendrá un nombre que nos servirá para podernos referir a ella) y escribir consistirá en guardar un valor en otro campo de texto.

El fichero de ejemplo ejemplo6.fla nos permitirá ver como usar esa manera de obtener y escribir los valores. En el apartado actividades de esta etapa hay una breve explicación de cómo se leen y escriben.

Revisar el archivo ejemplo6.fla

If else

La primera estructura de control que vimos fue la estructura secuencial. Dicha estructura no tiene ninguna instrucción que la identifique y, por tanto, no es necesaria ninguna traducción para poder utilizarla en ActionScript.

La siguiente estructura fue la condicional que escribíamos como:

```
Si (condición) {
    Instrucciones
}
```

o

```
Si (condición ) {
    Instrucciones a realizar si se cumple la condición;
} sino {
    Instrucciones a realizar si no se cumple la condición;
}
```

La traducción de estas estructuras a ActionScript es sumamente sencilla:

```
if (condición) {
    Instrucciones
}
```

o

```
if (condición ) {
    Instrucciones a realizar si se cumple la condición;
} else {
    Instrucciones a realizar si no se cumple la condición;
}
```

Así pasamos de tener un **si** – **sino** a tener un **if** – **else**.

for, while

Pues si la traducción del si es sencilla la traducción del para y del mientras también lo es.

Recordemos la sintaxis del **para**:

```
para ( variable = valor_inicial; variable <= valor_final;
     variable = variable + incremento) {
     instrucciones;
}
```

y del mientras:

```
mientras (condición) {
    instrucciones;
}
```

¿Y las traducciones a ActionScript? Nada más fácil, substituimos **para** por **for** y **mientras** por **while** y ya lo tenemos. Quedaría:

```
for ( variable = valor_inicial; variable <= valor_final;
     variable = variable + incremento) {
     instrucciones;
}
while (condición) {
    instrucciones;
}
```

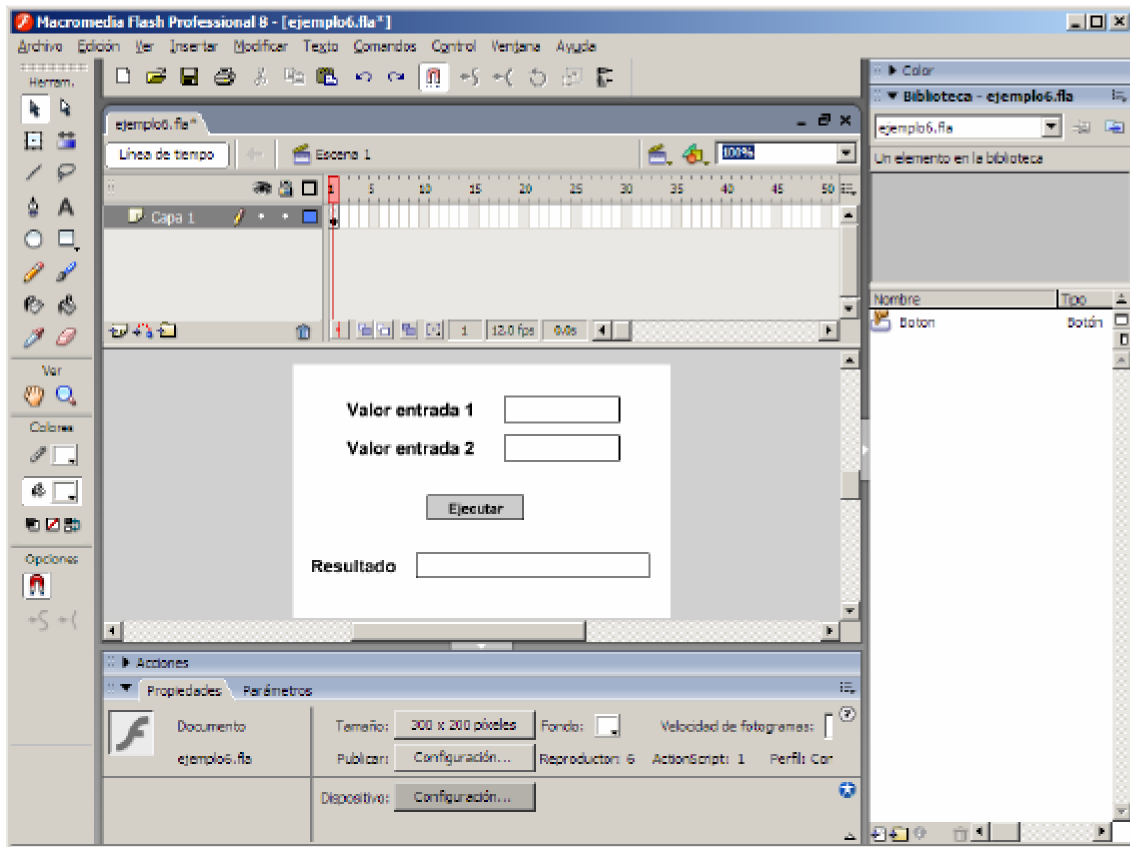
Ahora ya estamos en condiciones de escribir pequeños programas en ActionScript.

Actividades

El fichero ejemplo6.fla consta de dos entradas de texto, una salida y un botón ejecutar. Dentro de dicho botón podemos poner el programa que queramos que se ejecute. Así, muchos de los algoritmos que hemos ido haciendo a lo largo de este material podemos convertirlos en programas de una forma muy sencilla.

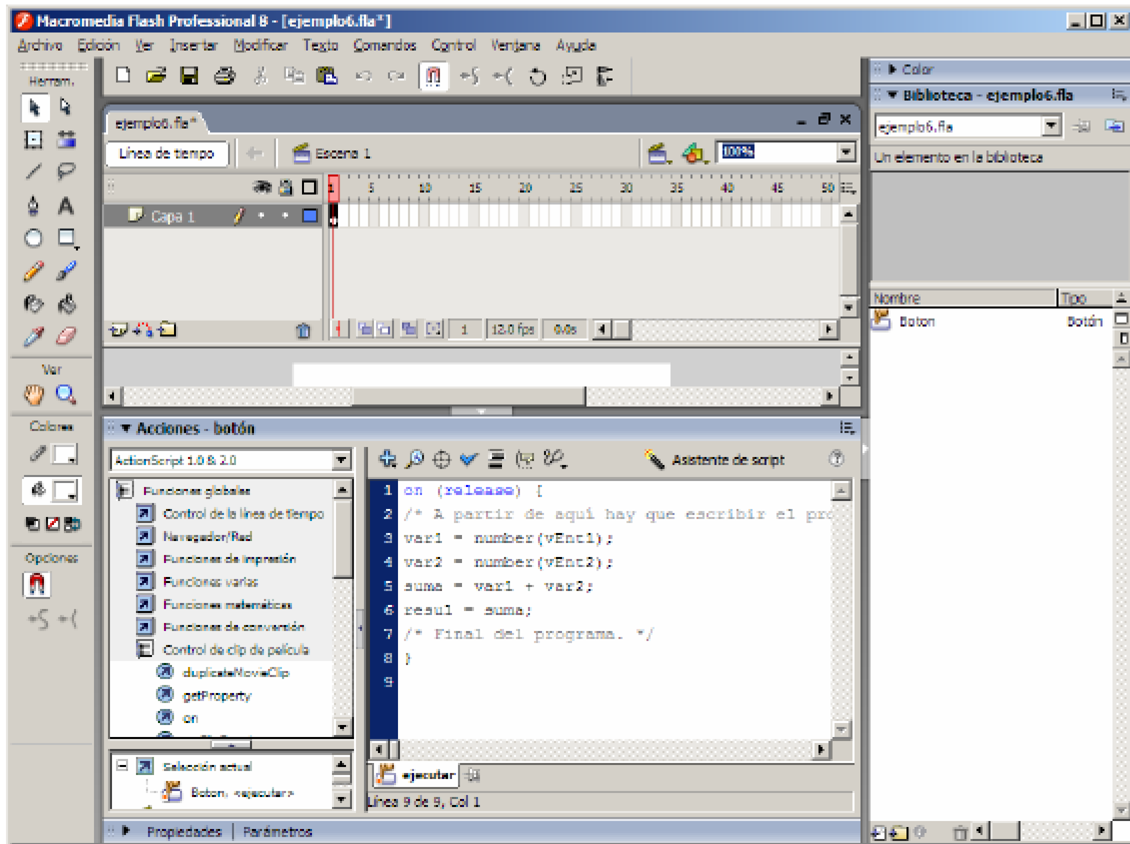
¿Cómo utilizar este fichero?

Primero lo abrimos con Flash



Después seleccionamos el botón de Ejecutar y hacemos un clic en la barra > Acciones y otro sobre la barra > Propiedades.

La pantalla quedará así:



Ya estamos en condiciones de escribir nuestro programa. Sólo un par de cosas a tener en cuenta. El archivo está creado pensando en programas que tengan una o dos entradas y una salida. Si el programa sólo necesita una entrada la otra no la usaremos y ya está.

Para guardar el valor de uno de los dos cuadros de texto de entrada en una variable hay que escribir

```
nombreDeLaVariable = vEnt1;
```

Sin embargo esto hará que el valor siempre sea de tipo texto, aunque hayamos introducido un número. Si lo que deseamos es que el valor que se guarde en la variable sea un número, la instrucción deberemos escribirla de la siguiente manera:

```
nombreDeLaVariable = Number(vEnt1);
```

Evidentemente, si nos interesa guardar un valor introducido en el segundo cuadro de texto no tendremos más que cambiar el 1 por un 2.

Para escribir el resultado en pantalla usaremos el cuadro de texto llamado resultado. Para guardar un valor en dicho cuadro de texto deberemos escribir:

```
resul = nombreDeLaVariable;
```

En el fichero ejemplo6.fla ya hay un pequeño programa creado para que sirva de ejemplo. Simplemente lee los valores introducidos en los cuadros de texto de entrada y escribe la suma de esos valores en el cuadro de texto resultado.

Revisar el archivo ejemplo6.fla

A partir de aquí escribir los siguientes programas en ese archivo:

1. Un programa que calcule la división de dos números
2. Un programa que dados dos números nos diga cual es el mayor.
3. Un programa que dado el nombre del usuario escriba “Hola nombre” (donde se supone que nombre es el nombre que el usuario ha introducido.
4. Pasar el algoritmo que calcula si un número es primo o no a un programa.
5. Un programa que dados dos números nos calcule el resultado de elevar el primero al segundo.

Otras estructuras

Introducción

Este apartado del material pretende dar al lector más avanzado algunas pautas para poder hacer programas más grandes y complejos. Hasta ahora, hemos vistos las estructuras básicas de programación, ahora iremos un poco más allá. Es muy importante no entrar en estos temas si no se tienen perfectamente asumidos los anteriores, el material que viene a continuación va más allá de los objetivos de este documento, pero, sin embargo se incluye para que el lector pueda, una vez acabado el tutorial, profundizar en el arte de la programación.

Hay dos conjuntos de temas en este apartado. Por una parte seguimos con la introducción a la programación con dos temas sumamente útiles como los conceptos de búsqueda y recorrido y las funciones y procedimientos (subprogramas). La otra parte es la explicación de cómo se hacen algunas de las cosas explicadas en este material en Flash.

En concreto, los temas que se tratarán son:

- Búsqueda y recorrido. Un par de conceptos que facilitan la elección de las condiciones de un bucle.
- Funciones y procedimientos. Subprogramas.
- Variables en Flash.
- Declaración de funciones en Flash.
- Objetos, métodos y propiedades en Flash.

Búsqueda y recorrido

Hasta ahora hemos visto como hacer bucles usando las instrucciones mientras y para. Sin embargo no hemos visto en que ocasiones es conveniente usar estas instrucciones, aunque con los ejemplos y los ejercicios, seguramente habrá quedado bastante claro.

Sin embargo, en muchas ocasiones, aquello que queremos hacer con un bucle se puede considerar como una búsqueda o un recorrido y, por tanto, es interesante conocer como se programan cada una de estas estructuras para, llegado el momento, hacer nuestro programa de la manera más simple posible.

Recorrido

Comenzamos con el recorrido porque es la estructura más simple. Hacemos un recorrido cuando nos interesa ir desde el principio hasta el fin. El ejemplo más claro sería, dado un conjunto de números, mirarlos todos. ¿Cuándo nos puede interesar eso? Pues cuando los queramos sumar todos, por ejemplo.

La característica principal del recorrido es que acabamos cuando se acaben todos los datos. Así, podríamos definir el recorrido como algo parecido a esto:

```
Mientras (nos queden datos por mirar) {
      Tratamos los datos;
}
```

En este caso, el *mientras* tiene una sola condición de final, que se acaben los datos. Veamos ahora un ejemplo:

Programa que, dado un valor inicial y otro final nos calcule la suma de todos los números entre los valores dados (incluidos).

Podríamos hacer este programa:

```
{
    leer (valorInicial);
    leer (valorFinal);
        suma = 0;
        mientras (valorIncial <= valorFinal) {
            suma = suma + valorInicial;
            valorInicial = valorInicial + 1;
        }
    escribir (suma);
}
```

Pero, en general, en un recorrido el mientras puede substituirse con un Para:

```
Para ( i=valorInicial ; i <= valorFinal ; i = i +1 ) {
    suma = suma + i;
}
```

Otros ejemplos de programas que pueden resolverse con un recorrido serían:

- Contar el número de letras que tiene una cadena de caracteres.
- Calcular todos los divisores de un número.
- Revisar el interés de todos los clientes de un banco.
- Dado un conjunto de números, buscar el mayor de ellos.

Búsqueda

La búsqueda es una estructura un poco más complicada, aunque no demasiado. Se llama así porqué es lo que hacemos cuando buscamos algo. Un caso simple sería, dado un conjunto de números, buscar si hay algún 0. En este caso, dado que sólo necesitamos saber si hay alguno, el bucle se acabará cuando aparezca un 0. Pero, ¿y si no hay ninguno? Tendremos entonces que controlar también que no se acaben los números.

Así, mientras que en el recorrido el mientras tenía una sola condición (que hubiésemos llegado al final) en la búsqueda tenemos dos condiciones posibles, que hayamos llegado al final y que hayamos encontrado el elemento que buscábamos.

A partir de esta información, podemos definir la búsqueda como:

```
Mientras (no_encontremos_el_elemento_que_buscamos y
queden_datos_por_mirar) {
    Tratamos los datos;
}
si (hemos_encontrado_el_elemento) {
    instrucciones;
} sino {
    instrucciones;
}
```

A diferencia del recorrido, donde una vez acabado ya teníamos el resultado sin más, en el caso de la búsqueda, una vez acabado el bucle es necesario comprobar si hemos encontrado el elemento que buscábamos o no, de ahí que tengamos la condición final.

Otros ejemplos de programas que pueden resolverse con una búsqueda serían:

- Buscar si en una cadena de caracteres hay algún punto.
- Dado un número decir si es primo o no.

- Dado un conjunto de números, decir si hay algún cero entre ellos.

Subprogramas: Funciones y procedimientos

En muchas ocasiones, cuando hagamos un programa, nos encontraremos con que hay trozos de programa que necesitamos repetir en varios sitios distintos. Por ejemplo, si tenemos un programa que calcule probabilidades seguramente necesitaremos calcular el factorial de un número en varias ocasiones. Lo ideal, en este caso, es poder tener el código necesario para calcular el factorial de un número una sola vez y usarlo cada vez que necesitemos el cálculo de un factorial. Pero, ¿Cómo se hace eso?

Todos los lenguajes de programación permiten la creación de subprogramas que pueden ser llamados desde cualquier punto del programa principal. De esta manera se pueden agrupar conjuntos de instrucciones que se repitan a lo largo del programa y que así sólo necesitamos escribirlas una vez. Además, como a los subprogramas les pondremos un nombre, podremos usarlos para ordenar mejor el programa y que sea más comprensible.

En general estos subprogramas los podemos dividir en dos tipos. Aquellos que devuelven un resultado y aquellos que no devuelven ningún resultado. A los que devuelven un resultado los llamamos funciones y a los que no lo devuelven les llamamos procedimientos. Un ejemplo de una función sería el cálculo del factorial, en que, dado un número, la función devuelve el factorial de ese número. Un caso de un procedimiento sería un subprograma que escriba unos resultados en la pantalla de una determinada manera.

Generalizando, los procedimientos los podemos definir así:

```
procedimiento (parámetros) {
    instrucciones;
}
```

Donde los parámetros son un conjunto de variables donde guardaremos los valores que nos interesa que el programa haga llegar al subprograma (por ejemplo los valores que queremos que se escriban en la pantalla).

Y una función la podemos definir así:

```
función (parámetros) {
    instrucciones;
    retorna expresión;
}
```

Dónde los parámetros son lo mismo que en los procedimientos y la expresión de retorno será el valor que queremos que retorne la función.

Parámetros

Una parte importante de la declaración y utilización de funciones y procedimientos radica en el paso de parámetros. Los parámetros son los datos que recibirá el subprograma para trabajar. Para cada dato que pasemos usaremos una variable, separando los diferentes parámetros con comas. Así, por ejemplo si hacemos un procedimiento que escriba varios datos en la pantalla, podríamos hacer lo siguiente:

```
procedimiento pantalla (nombre, apellidos, edad) {
    escribir ("La edad de ");
    escribir (nombre + " " + apellidos);
    escribir (" es de ");
    escribir (edad + " años");
```

```
}
```

En este ejemplo se reciben 3 parámetros que se convierten en tres variables que se usan dentro del procedimiento.

En una función la situación es similar

```
función cuadrado (x) {
    x = x * x;
        retorna x;
}
```

Lo mejor de esto es que esta función (o el procedimiento que hemos descrito anteriormente), la podemos usar cuando nos vaya bien. Así, por ejemplo, en un programa podríamos tener algo como esto:

```
a = 5;
b = 7;
f = cuadrado (a) + cuadrado (b);
```

Cuando llamamos a la función cuadrado la primera vez, le enviamos como parámetro un 5 (el valor que almacena la variable a) y cuando lo hacemos la segunda le enviamos un 7 (el valor que almacena la variable b). Así en f guardaremos el valor 74 (25 + 49).

Paso de parámetros: Paso por valor y paso por referencia

El paso de parámetros a un subprograma podemos hacerlo de dos maneras diferentes, por valor y por referencia.

El paso por valor es el más sencillo. Se pasa un valor. Así, en el ejemplo de la función cuadrado es como si en vez de poner

```
f = cuadrado (a) + cuadrado (b);
```

Se pusiese

```
f = cuadrado(5) + cuadrado(7)
```

Lo que recibirá la función cuadrado será en un caso 5 y en el otro un 7.

Sin embargo el paso por referencia es un poco más complicado. Lo que se pasa es una referencia a la variable inicial. Así, cuando ponemos (suponiendo que el parámetro se pasa por referencia)

```
cuadrado(a)
```

lo que pasamos es una referencia a la variable a. Es como si pasásemos la variable y no su valor. Así, mientras que en el paso por valor x era una nueva variable, en el paso por referencia x es un alias de la variable a, y, por tanto, cualquier modificación de la variable x afectaría a la variable a.

Vamos a verlo con un ejemplo.

```
a = 5;
b = 7;
f = cuadrado (a) + cuadrado (b);
escribir (a);
escribir (b);
```

Si el paso de parámetros es por valor, en pantalla se presentarían los valores 5 y 7, mientras que si el paso es por referencia los valores que se escribirán en pantalla serán 25 y 49 (lo que modificamos en x se ha modificado en a o b).

¿Cómo sabemos si el paso de parámetros es por valor o por referencia? Cada lenguaje utiliza el paso de parámetros de una manera diferente, aunque en un lenguaje que permita escoger, lo habitual será especificarlo en la definición del subprograma. Siguiendo con el ejemplo del cuadrado, y siguiendo la notación similar a la que se usa en el lenguaje Pascal, el paso de parámetros sería por valor en este caso:

```
función cuadrado (x) {
}
```

Y por referencia en este otro:

```
función cuadrado (var x) {
}
```

¿Qué utilidad tienen los diferentes pasos de parámetros?

Habitualmente usaremos como paso de parámetros el paso por valor. De esta manera podemos hacer nuestros subprogramas más independientes del programa donde los usemos. Sin embargo dado que una función sólo permite retornar un valor, cuando necesitemos devolver más, deberemos usar un parámetro por referencia para usarlo como segundo (o enésimo) valor de retorno.

Variables en Flash

Flash tiene, básicamente, tres tipos de datos diferentes: de cadena, numéricos y booleanos (String, Number y Boolean) y podemos usarlos indistintamente sin necesidad de declararlos ni definir nada. Sin embargo, por claridad, siempre es conveniente definir las variables e indicar el tipo de datos que queremos guardar en ellas.

Así, si queremos usar una variable de tipo numérico la definiremos de la siguiente manera:

```
id_variable = new Number;
```

O, más práctico,

```
id_variable = 5;
```

En cualquiera de los dos casos estamos haciendo lo mismo, sólo que en el segundo caso, además, le damos un valor a la variable. De todas formas, en realidad, en Flash no importa el sistema que utilicemos, ya que en cualquier momento podemos dar cualquier valor a una variable, incluso de un tipo diferente al que usamos al declararla. Sin embargo, la declaración de variables nos facilitará la lectura e interpretación del código que escribamos, así que siempre será mejor usarla.

Variables, variables locales y variables globales

En Flash podemos definir el ámbito de visibilidad de las variables (donde se "ven" y por tanto se pueden usar) de dos maneras diferentes. Por una parte, usando la palabra clave var, podemos definir variables locales. Además, usando el objeto _global, podemos definir variables globales.

Una variable local es aquella que sólo es visible (sólo se puede usar) en el entorno en el que ha sido declarada, por ejemplo, una función. Una variable global, en cambio, es visible desde todas partes. ¿Y el resto de variables? Dependerá del momento en que hayan sido declaradas. Si han sido declaradas en un procedimiento desde el cual se llama a otro, será visible en ambos. En general, es conveniente usar siempre variables locales (los parámetros se

comportan como variables locales) y especificar como globales aquellas que sea necesario. Cuando usamos variables locales no tenemos que preocuparnos de si esa variable se usa en algún otro sitio y, de esta manera, garantizamos que no sobrescribimos alguna variable definida en algún otra función.

Veamos tres ejemplos de variables:

```
var a = 5;
b = 7;
_global.c = 9;
```

Aquí tenemos una variable local (a) y una variable global (_global.c) Hay que tener presente que para referirnos a la variable global siempre deberemos usar el _global antes.

Finalmente, en b, como no hemos definido su ámbito de actuación, sera visible en el entorno en que la hemos creado, pero, además, también será visible en las funciones que llamemos desde dicho entorno.

Funciones en Flash

Como en otros lenguajes de programación, en Flash los subprogramas son siempre funciones, no existen, como tales, los procedimientos. La declaración de una función la haremos de la siguiente manera:

```
function id_funcion (parámetros) : tipo_retorno {
    instrucciones;
    return expresión;
}
```

Los parámetros son opcionales, así como el tipo de retorno. Si no se ponen los parámetros, cuando se llame a la función se tiene que poner con los paréntesis (aunque no tengan nada dentro). En cuanto al tipo de retorno, indica el tipo del dato que retornará la función. Si se pone, la instrucción return es obligatoria. Si no se especifica tipo de retorno, la instrucción return se puede poner o no (pero si no se pone no retornará nada) y si se especifica como tipo de retorno void, entonces la función se comporta como un procedimiento.

Los parámetros de una función se comportan como variables locales dentro de ella. Además, si declaramos variables con la instrucción var delante, también serán consideradas locales.

Como ya se ha comentado al hablar de las variables, es muy importante que todas las variables que se usen en una función sean declaradas como locales. De esta manera evitamos conflictos con variables definidas en otras funciones o procedimientos.

En cuanto al paso de parámetros hemos de tener presente que, en Flash, el paso de parámetros depende de si la variable guarda un valor de un tipo simple (Number, Boolean o String) o si guarda un objeto (tipo Object).

Si la variable guarda un valor de un tipo simple el paso de parámetros se hace siempre por valor sin que Flash ofrezca ninguna posibilidad de hacer el paso por referencia. En cambio, si la variable guarda un objeto el paso de parámetros siempre es por referencia.

Objetos, métodos y propiedades en Flash

ActionScript es un lenguaje de programación que en algunos aspectos se acerca mucho a lo que es la orientación a objetos. De hecho, en ActionScript 2.0, podemos definir clases tal y como lo haríamos en otros lenguajes orientados a objetos.

Aunque no vamos a ver aquí que es la orientación a objetos y como funciona, si que vamos a ver sus “consecuencias” en la utilización de algunas funciones, así como la utilidad que puede tener para nosotros el tipo Object que nos permite generar objetos a partir de la definición de sus propiedades y sin la definición de una clase previa.

En un objeto (en cualquier lenguaje de programación orientado a objetos) encontramos métodos y propiedades. En las propiedades almacenamos los valores que asignamos al objeto y los métodos son las funciones que nos permiten manipular esos datos.

En ActionScript tenemos una pseudoclase que nos es muy útil para trabajar con cadenas de caracteres: la clase String. Vamos a ver un método y una propiedad de esta pseudoclase y como podemos usarlos.

Para crear un objeto de tipo String usamos la instrucción new:

```
a = new String;
b = new String (“Hola”);
```

En el primer caso hemos creado el objeto a con una cadena vacía y en el segundo caso hemos creado un nuevo objeto pero con un contenido.

Podemos decir que los objetos de tipo String tienen dos propiedades, la cadena de caracteres que contienen y el número de caracteres que tiene esa cadena. Para acceder a la cadena de caracteres lo haremos usando directamente el nombre de la variable, pero para acceder al número de caracteres, deberemos usar la propiedad length.

Así, si ponemos x = b.length en x estaremos almacenando el valor 4 (de Hola).

Por otra parte, la clase String tiene algunos métodos que nos pueden ser de utilidad. Uno de ellos es substr:

```
my_str.substr (inicio, [tamaño])
```

Retorna una subcadena de my_substr iniciandola en inicio y tomando tantos caracteres como indique tamaño. Si no se especifica el tamaño, toma toda la subcadena a partir de la posición de inicio.

Ej.: x = b.substr(1,2) guardaría en x la cadena “ol” porque las cadenas de caracteres empiezan a contarse a partir de la posición 0 y, por tanto, en la posición 1 está la o.

La pseudoclase String se comporta en algunas cosas como una clase (tiene métodos y propiedades) y en otras como un tipo (en el paso de parámetros se pasa siempre por valor).

El tipo Object

Un tipo que nos puede ser de mucha utilidad en algunos casos es el tipo Object. Con este tipo de datos podemos crear objetos con sus correspondientes propiedades, de manera que podamos guardar en una sola variable un conjunto de datos relacionados.

Por ejemplo, podríamos definir:

```
Persona = new Object;
Persona.nombre = “Diego”;
Persona.apellidos = “Martínez Casanova”;
Persona.edad = 22;
```

De esta manera tenemos todos los datos de una persona juntos en una sola variable. Este tipo de datos, tiene además una característica que en algunos casos será una ventaja y en otros un inconveniente y es que siempre se pasa por referencia.

Ejercicios

Búsqueda y recorrido

Antes de hacer los siguientes ejercicios es recomendable pensar cómo lo resolverías: ¿Cómo un recorrido o cómo una búsqueda?

1. Dado un número escribir 2 de sus divisores.
2. Dado un número escribir el mayor de sus divisores.
3. Dados dos números escribir la suma de todos los que están entre ellos.
4. Dado un número decir si es primo.
5. Dado un número contar cuantos divisores tiene.
6. Dados dos números, contar cuantos divisores tiene el primero, que sean más pequeños que el segundo.
7. Dada una cadena de caracteres, contar cuantas veces aparece la letra a.
8. Dada una cadena de caracteres, decir si tiene alguna ñ.
9. Dada una cadena de caracteres, eliminar todos los espacios en blanco.
10. Dada una cadena de caracteres, contar cuantos espacios hay antes de la primera letra.
11. Dada una cadena de caracteres, obtener la primera palabra.

Funciones

12. Haz una función de reciba un número y retorne cierto o falso según sea primo o no.
13. Haz una función que reciba dos números (a y b) y retorne a elevado a b.
14. Haz una función que reciba un número y retorne el factorial de ese número. El factorial de un número n se obtiene multiplicando todos los números menores o iguales que el. Es decir:
    n! = n * n-1 * n-2 * n-3 * ... 3 * 2 * 1
15. Haz una función que reciba una cadena y una letra y retorne el número de veces que aparece esa letra en la cadena.
16. Haz una función que reciba una cadena y retorne la misma cadena sin espacios.
17. Haz una función que reciba tres números y retorne el mayor.
18. Haz una función que dado un número devuelva el número de divisores que tiene.

Una vez hechas esas funciones, puedes hacer estos programas que las usen:

19. Hacer un programa que dados dos números calcule la fórmula combinatoria n! / (m! * (n-m)!) Nota: n! es factorial de n. Usa la función que has definido antes para hacer los cálculos.
20. Dado un número mayor que 2, escribir los dos números primos más grandes, que sean más pequeños que el número dado (si el número dado es 10, los dos primos más grandes serán 5 y 7). Utiliza la función que calculaba si un número era primo o no.
21. Dadas dos cadenas de caracteres, decir cual de las dos tiene más a. Utiliza la función que hiciste en el apartado 4 de funciones para hacer este ejercicio.

Programación

Aprovechando el fichero de ejemplo ejemplo7.fla haz en ActionScript los ejercicios anteriores.

El programa deberás hacerlo en la función *programa*, y debes usar como datos de entrada las variables v1 y v2 y como datos de salida las variables r1 y r2. Modifica el fichero y adáptalo a tus necesidades si deseas algún ejercicio diferente a los que te proponemos aquí.

Ten presente que las variables v1 y v2 son de tipo String. Si el dato de entrada es un número y queremos usarlo, deberemos definir una nueva variable y guardar en ella Number(v1) (nv1 = Number (v1);)

Ejemplos

En el archivo ejemplo71.fla está el ejercicio 20 solucionado y en el archivo ejemplo72.fla está el ejercicio 9 solucionado. En caso de que tengas dudas, puede ser una ayuda el consultar estos ejemplos.

Revisar los archivos ejemplo7.fla, ejemplo71.fla y ejemplo72.fla