Programación en JavaScript/Texto completo

< Programación en JavaScript

Programación en JavaScript

por es.wikibooks.org

Sumario

Introducción

El JavaScript es un lenguaje de programación ampliamente utilizado en el mundo del desarrollo web por ser muy versátil y potente, tanto para la realización de pequeñas tareas como para la gestión de complejas aplicaciones. Además, ha saltado el ámbito de la web, y también podemos encontrarlo en otros entornos, como es el lenguaje ActionScript de Macromedia, que acompaña al sistema Flash.

JavaScript es un lenguaje interpretado que, al contrario de las aplicaciones normales, que son ejecutadas por el sistema operativo, es ejecutado por el navegador que utilizamos para ver las páginas. Eso hace que podamos desarrollar aplicaciones de muy diverso tipo, desde generadores de HTML, comprobadores de formularios, etc..., hasta programas que gestionen las capas de una página. Pueden desarrollarse incluso aplicaciones que permitan poder tener capas en una página como si fueran ventanas, y dar la sensación de estar trabajando con una aplicación con interfaz de ventanas.

JavaScript no es el único lenguaje que podemos encontrar en la web. También tenemos a su gran contrincante: Visual Basic Script. Este lenguaje, desarrollado por Microsoft, está basado en el lenguaje Basic de esta empresa, pero desgraciadamente sólo puede utilizarse en el navegador Internet Explorer. Si queremos que nuestras páginas dinámicas puedan ser vistas desde cualquier navegador y sistema operativo, la elección más adecuada es sin lugar a dudas JavaScript.

¿Qué podemos hacer con JavaScript?

- Páginas dinámicas (DHTML)
- Comprobación de datos (Formularios)
- Uso de los elementos de la página web
- Intercambiar información entre páginas web en distintas ventanas
- Manipulación de gráficos, texto, etc...
- Comunicación con plug-ins: Flash, Java, Shockwave, etc...

Características de JavaScript

JavaScript comparte muchos elementos con otros lenguajes de alto nivel. Hay que tener en cuenta que este lenguaje es muy semejante a otros como C Java o PHP, tanto en su formato como en su sintaxis, aunque por supuesto tiene sus propias características definitorias.

JavaScript es un lenguaje que diferencia entre mayúsculas y minúsculas, por lo que si escribimos alguna expresión en minúsculas, deberemos mantener esa expresión en minúsculas a lo largo de todo el programa. Si escribimos esa misma expresión en mayúsculas, será una expresión diferente a la primera. Esto es así en la mayoría de los lenguajes de este tipo, como PHP

Otra característica es que podemos encerrar las expresiones que escribamos con una serie de caracteres especiales. Estos caracteres se denominan operadores y sirven tanto para encerrar expresiones como para realizar trabajos con ellas, como operaciones matemáticas o de texto. Los operadores que permiten encerrar expresiones deben abrirse siempre. '(', '{' y '[' y deben cerrarse con sus correspondientes ')', '}' y ']', respectivamente.

Como JavaScript es un lenguaje de formato libre, podemos escribir las líneas de código de la forma que consideremos mejor, aunque por supuesto debemos escribir siempre de la forma correcta. Por ejemplo, podemos escribir las líneas con un número variable de espacios:

```
variable = "hola";
variable="hola";
    variable ="hola";
variable= "hola"   ;
```

Esto significa que podemos añadir tabuladores al inicio de la línea para justificar los párrafos de código. También podemos romper las líneas de texto si son demasiado largas:

```
document.write("Muy \
buenas");
```

Pero no podemos hacer esto:

```
document.write \
("Muy buenas");
```

Sólo podemos romper cadenas de texto, no instrucciones.

En ocasiones tenemos que escribir algunos caracteres especiales llamados "de escape", porque en ocasiones hay que utilizar algunos caracteres no imprimibles, como por ejemplo:

```
document.write ("Tú y yo somos \"programadores\".");
```

Podemos ver que al introducir comillas dobles dentro de la cadena de caracteres, debemos añadir la barra invertida '\' para escapar las propias comillas, aunque esta misma expresión se podía haber escrito así:

```
document.write ("Tú y yo somos 'programadores'.");
```

Otro aspecto importante de JavaScript es la necesidad o no de utilizar el punto y coma ';' al final de las instrucciones. Este operador sólo sirve para delimitar las instrucciones, pero aunque su uso es obligatorio en la mayoría de los lenguajes, es opcional en JavaScript. Si usamos este operador, podemos incluir varias sentencias en la misma línea de código, y si no lo usamos, sólo podemos escribir una sentencia cada vez. De todas formas, aconsejamos usar el punto y coma porque en otros lenguajes como PHP o Java, este operador es obligatorio.

Las variables en JavaScript

En un programa JavaScript nos podemos encontrar con dos elementos básicos: **código** y **datos**. La parte del código es la que nos permite hacer cosas dentro de la página web, y la parte de datos es la que define el estado de la página web en un momento determinado. Los datos de un programa se guardan dentro de **variables**.

¿Qué es una variable? Una variable es como una caja: dentro de una caja nosotros podemos guardar cosas. Sólo que en las variables de JavaScript la caja sólo puede guardar una sola cosa a la vez. ¿Y por qué se las llama variables? Se las denomina así porque su contenido puede cambiar en cualquier momento durante el desarrollo del programa. De esta forma, una variable en JavaScript puede contener distintas cosas según donde se encuentre en el programa: números, letras, etc...

Tipos de datos en JavaScript

Existen cuatro tipos básicos:

- **Números** (enteros, decimales, etc.).
- **Cadenas de caracteres**
- **Valores lógicos** (True y False).
- **Objetos** (una ventana, un texto, un formulario, etc.).

Declaración de variables

A continuación aparece un ejemplo de declaración de una variable en JavaScript:

```
var miVar = 1234;
```

Aquí hemos definido una variable numérica con un valor entero. Pero también podríamos definir un número con decimales:

```
var miVar = 12.34;
```

Como podemos ver, la nueva variable usa el operador '.' (punto) para distinguir entre la parte entera y la parte decimal. Esto es importante, porque no podemos usar la coma como hacemos en España.

Si queremos definir una cadena de caracteres, lo podemos hacer de la siguiente manera:

```
var miCadena = 'Hola, mundo';
// o bien:
var miCadena = "Hola, mundo";
```

Aquí vemos que podemos usar los dos tipos de comillas para crear cadenas de caracteres, y será muy útil cuando trabajemos con ello (podemos incluso combinarlas dentro de la misma cadena).

También podemos crear variables con valores lógicos. Eso significa que la variable podrá tener sólo dos valores: verdad o mentira.

```
var miVar = true;
```

Los valores admitidos para este tipo de variables son *true* y *false*. Este tipo de variables nos vendrán muy bien para crear condiciones y como valor para devolver en funciones, que veremos más adelante.

Y por último tenemos un tipo de dato especial: los objetos. ¿Y qué son los objetos? Son "cosas" que podemos usar en nuestro programa para representar "entidades". Esto lo entenderemos muy fácilmente con unos ejemplos.

Estamos rodeados de objetos: mesas, libros, monitores, ratones, cuadros, etc... Algunos son más simples y otros son más complicados. Podemos manipular todos ellos según sus características y su forma de interactuar con el entorno donde están. Por ejemplo, una mesa sabemos que tiene cuatro patas, una tabla lisa, y que es de un color o varios colores. Es decir, que podemos determinar una mesa por sus propiedades o atributos. Pero además, con la mesa podemos hacer cosas: podemos poner cosas encima, podemos usarla para comer o leer y a veces podemos colgar cosas de ellas, por ejemplo en un revistero. Todo eso son métodos o comportamientos que la mesa tiene para interactuar con el resto de su entorno.

Pues bien, podemos decir que los objetos en JavaScript son muy parecidos: tienen **propiedades** (datos) y **métodos** (código). Podemos crear y usar objetos para manejar elementos del navegador web: una ventana del navegador es un objeto *window*, una página HTML es un objeto *document*, y una imagen es un objeto de tipo *Image*. Es fácil darse cuenta de que los objetos son de un determinado tipo: un objeto mesa, un objeto ventana, un objeto ratón, etc... Todos los objetos de un mismo tipo tienen características semejantes, aunque luego cada objeto tiene propiedades con valores distintos dependiendo de cada caso. Así, dos mesas puede tener color marrón o azul, pero las dos seguirán teniendo patas, que pueden ser 4 ó 5, depende... En JavaScript, los objetos son muy importantes, como vamos a comprobar en el siguiente capítulo, que trata de los arrays (matrices) y las sentencias de control.

Operar con variables

Como ya estudiamos en el capítulo anterior, en JavaScript podemos definir unos elementos llamados variables que nos permiten almacenar datos de distintos tipos. Naturalmente, nosotros podemos usar esos datos en nuestros programas y, como ya se indicó entonces, podemos incluso variar esos datos manteniendo la variable donde los depositamos. A esta operación se le llama modificar la variable, y es una de las bases de la programación moderna.

Las variables podemos usarlas en multitud de situaciones, al mostrar datos, al enviarlos y recibirlos, en expresiones y llamadas a funciones... Podemc tratar con variables para almacenar los datos que vamos a usar a lo largo del programa, tanto globalmente en toda la aplicacion como de forma exclusiva con las funciones que creemos, como veremos en el capítulo correspondiente.

```
var numero = 1;
numero = numero + 2;
numero += 3;
```

El resultado final de esta operación sera que la variable numero será igual a 6. En la primera línea lo que hemos hecho es declarar la variable numero con el valor inicial 1. Despues, hemos incrementado el valor de la variable con la misma variable, sumándole 2, y posteriormente hemos vuelto a incrementar la variable sumándole 3 por medio del operador tipográfico +=. Los operadores se encuentran en el primer apéndice del curso.

Sin embargo, surge un pequeño problema cuando tenemos que tratar con cantidades mayores de datos. Las variables como tales sólo nos permiten gestionar un dato cada una de ellas, con lo que cuando tenemos que gestionar grupos mayores de datos, se hace realmente complicado. Miremos el siguiente ejemplo, en el que definimos unos nombres:

```
var nombre1 = 'pepe';
var nombre2 = 'toño';
var nombre3 = 'mari';
var nombre4 = 'lucas';
var nombre5 = 'sonia';
var nombre6 = 'ruth';
var nombre7 = 'tete';
```

Si ahora quisiéramos listar estos datos (más adelante veremos cómo), tendríamos que referirnos a cada variable en concreto, con lo que tenemos pululando por nuestro programa siete variables a las que será difícil referirnos de una forma genérica (por ejemplo, como estudiaremos más adelante, para listarlos dinámicamente en un formulario). Para resolver este problema tenemos una solución: los arrays (matrices).

Arrays (Matrices)

Las matrices son variables que contienen un objeto de tipo *Array*. Podemos definir una matriz de la siguiente manera:

```
var matriz = new Array();
```

De esta forma, hemos creado una matriz vacía que puede contener un numero ilimitado de elementos, tantos como nos permita el sistema donde se ejecuta. Las matrices vienen a ser como cajas que en vez de contener una sola cosa, contienen muchas, como si pudiéramos dividir la caja en compartimentos en los cuales pudiéramos ir depositando cosas.

Además, podemos crear matrices con una "dimensión", es decir, que podemos hacer que la matriz se inicie con un número de elementos determinado:

```
var matriz = new Array(15);
```

Con esta instrucción, lo que hemos hecho es crear una matriz de quince elementos. Pero ahora lo interesante es saber cómo llamar a esos elementos, ya que si creamos la matriz, pero no sabemos operar con ella, no sirve para mucho, ¿no? La forma de acceder a un elemento de la matriz es la siguiente:

```
elemento = matriz[1];
```

En este ejemplo, la variable *elemento* contendrá el valor del elemento 1 de la matriz. Es lo que se llama **índice** de la matriz, e identifica a cualquiera de los elementos de la matriz. Hay que fijarse en que utilizamos los corchetes "[]" para señalar un elemento de la matriz. El primer elemento de la matriz es el de índice ‘0’, no el de índice ‘1’. Así, para el anterior ejemplo de una matriz de 15 elementos, el último elemento posible es el 14.

De la misma forma, podemos dar un valor a cualquiera de los elementos de la matriz:

```
matriz[5] = ‘hola’;
```

Hemos asignado el valor *hola* al elemento 5 de la matriz. Los elementos de una matriz pueden contener cualquier tipo de dato, y se pueden cambiar en cualquier parte del programa, al igual que ocurre con las variables.

¿Y si queremos saber cuántos datos tenemos en la matriz? Como dijimos antes, las matrices son objetos de tipo Array, y los objetos pueden tener atributos (datos) y funciones (código). El atributo que debemos usar con matrices es **length**:

```
longitud = matriz.length;
```

De esta forma, podemos saber cuantos elementos tiene la matriz. Recordad que como el primer índice es ‘0’, el último elemento será siempre *matriz.length - 1*.

Si necesitamos que la matriz contenga más elementos, podemos redimensionar la matriz aplicándole un nuevo objeto de matriz:

```
matriz = new Array( longitud que queramos );
```

Sin embargo, perderemos todos los elementos que tuviéramos anteriormente.

Sentencias de control

Una vez que nos sabemos manejar con variables y matrices, que son los datos de nuestros programas, debemos aprender a crear código, que es lo que hace que nuestro programa funcione.

Hasta ahora hemos visto cómo manejar los datos del programa, pero sólo con variables no podemos manejar un programa. Necesitamos escribir código fuente que nos permita conseguir que el programa haga cosas. Normalmente los programas tienen un flujo de ejecución: se ejecuta línea a línea, interpretándolas y comprobando el resultado de la ejecución. Pero a veces eso no es suficiente. A veces tenemos que controlar lo que hace el programa, permitiéndonos ejecutar un código u otro dependiendo de las circunstancias, o bien repetir el mismo código varias veces según unas condiciones concretas.

Cuando nos levantamos por la mañana, miramos por la ventana para decidir qué ropa nos ponemos. Decidimos entre una camisa blanca o un jersey azul dependiendo de si hace frío o no:

```
   Si hace frío -> me pongo el jersey.
   Si hace calor -> me pongo la camisa.
```

Condicionales

If

En un programa podemos realizar el mismo tipo de decisiones gracias a la instrucción **if**:

```
if (condición) {
   código_A
}
else {
   código_B
}
```

Veamos lo que tenemos aquí. Primero, comprobamos la condición en la que vamos a basar nuestra decisión (veremos más adelante cómo crear condiciones). Si la condición es válida (si es verdadera), se ejecutará el primer bloque de código que está entre llaves ({}); si no, se ejecutará el siguiente bloque de código, el que se encuentra debajo de la palabra *else*. Es decir, si la condición es verdadera, ejecutamos *código_A*, y si no, *código_B*.

De esta forma, podemos tomar decisiones simples que nos permiten cambiar el código que ejecuta el navegador, de forma que podemos controlar su ejecución fácilmente. Un ejemplo es este:

```
var edad = 18;
if (edad >= 18) {
   alert("Eres mayor de edad");
}
else {
   alert("Eres menor de edad");
}
```

Primero definimos la variable *edad* con el valor numérico 18. Después comprobamos si el sujeto es o no mayor de edad leyendo el valor de la variable: si *edad* es mayor o igual que 18, entonces le decimos que es mayor de edad; si no, le decimos que es menor de edad.

alert es una función de Javascript que nos permite mostrar un mensaje en pantalla.

Pero, ¿qué ocurre cuando queremos hacer varias comprobaciones al mismo tiempo? Entonces podemos usar la instrucción **else if**.

```
if (edad > 18) {
   alert("Tienes más de 18 años");
}
else if (edad == 18) {
   alert("Tienes 18 años");
}
```

```
else {
   alert("Tienes menos de 18 años");
}
```

De esta forma, podemos extender las condiciones todo lo que queramos para cubrir todos los casos necesarios. Hay que destacar que todas las condiciones son sucesivas y que sólo se ejecuta una de ellas. Además, el último *else* es el que se ejecuta en el caso de que ninguno de los *if* anteriores sea válido.

Switch

Hemos visto cómo gracias a *if-else* podemos decidir qué hacer en determinadas situaciones. Pero a veces sólo querremos decidir entre unos valores, y ejecutar código dependiendo de los posibles valores de una variable o devueltos por una función. Para no ensuciar tanto el codigo usando muchas sentencias *if* podemos usar la sentencia **switch-case**:

```
switch (variable) {
   case valor1:
      // codigo para valor 1
      break;
   case valor2:
      // codigo para valor 2
      break;
   case valor3:
   case valor4:
   case valor5:
      // codigo para valor 3, valor 4 y valor 5
      break;
   default:
      // código si no es ninguno de los anteriores
}
```

En este ejemplo, comprobamos varios valores de la variable *variable*, de forma que cada uno de ellos permite ejecutar un código diferente dependiendo de la situación. Ahora bien, si nos fijamos en *valor3*, *valor4* y *valor5*, comprobamos que sólo se ejecutará el código que aparece bajo *valor5*, porque los otros valores están vacíos, pero también porque no tienen una cláusula **break**. Esta palabra reservada corta la ejecución del código Javascript en el punto en que aparece, saliendo del bloque de código en el que se encuentra. Esto también nos servirá, por ejemplo, cuando veamos bucles.

Con esto, comprobamos que **switch** nos puede ayudar a ejecutar código de forma condicional, pero sin las complicadas operaciones que debemos realizar en *if*. Sin embargo, *switch* no puede nunca sustituir a *if*, situación que a la inversa sí ocurre.

Operadores lógicos y relacionales

Para crear condiciones, debemos usar dos elementos nuevos: los **operadores lógicos** y los **operadores relacionales**. Los operadores lógicos son 3:

Operador	Significado
&&	Y
||	O
!	NO

Y los relacionales son:

Operador	Significado
==	Equivalencia
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que
!=	Distinto de

Podemos crear condiciones simples usando los operadores relacionales, y después formar condiciones más complejas juntando otras más simples usando los operadores lógicos. Por ejemplo:

```
if (edad > 6 && edad < 12) alert("Eres un niño");
else if (edad >= 12 && edad < 18) alert("Eres un adolescente");
else if (edad >= 18 && edad < 31) alert("Eres un joven");
else if (edad >= 31 && edad < 60) alert("Eres un adulto");
```

```
else if (edad >= 60 alert ("Eres un adulto mayor");
else alert ("Tu debes de ser un bebé no mayor de 6 años");
```

Aquí hemos anidado varias condiciones para poder mostrar mensajes dependiendo de la situación en que nos encontremos. Podemos ver cómo es posible tomar decisiones más complejas que las que veíamos al principio del tema juntando varias condiciones por medio de los operadores lógicos. Debemos usar **&&** (*AND* lógico) cuando queramos que la resolución de la condición se encuentre incluida en las dos condiciones que indicamos. Debemos usar || (*OR* lógico) cuando queramos que la resolución de la condición no se incluya en las condiciones que usamos. Así, cuando decimos que *(edad >= 31 || edad <= 6)*, estamos indicando que el resultado ha de ser o bien mayor de 31 o menor de 6, pero nunca se encontrara en las dos condiciones a la vez, al contrario que con *&&* (que indica que se tienen que cumplir las dos condiciones a la vez para que sea verdad). De esta forma, podemos tomar decisiones que nos permitirán controlar el flujo del programa.

Bucles

En programación, la función principal de un programa es aquella que es ejecutada para que sus instrucciones sean realizadas, bien por el procesador central (en el caso de un lenguaje compilado como C o Pascal), o por un intérprete, que es un intermediario entre el programa y el procesador central. De esta forma, la ejecución lineal de las instrucciones genera acciones, cosas que nuestro programa hace porque se lo hemos indicado. Ahora bien, ¿Qué ocurre cuando queremos hacer varias veces una misma cosa? Tendríamos que escribir el mismo código varias veces, aunque el problema sería más peliagudo si encima el número de veces a ejecutar el código fuera variable, cambiase según el estado del programa (definido por las variables). Para ello tenemos dos tipos de elementos fundamentales: bucles y funciones.

Un bucle permite repetir varias veces el mismo código en la misma situación, incluso aunque la repetición de ese código sea variable y no la misma cantidad de veces cada vez. Como estudiaremos luego, tenemos a nuestra disposición dos bucles, for y while. En el caso de JavaScript, los bucles son condicionales, es decir que, como en el caso de if, necesitamos de condiciones como las que hemos estudiado para realizar las repeticiones de código. En otros lenguajes de programación existen bucles incondicionales, lo que significa que sólo pueden ir de cierto punto a cierto punto (por ejemplo, para contar de 1 a 10, aunque esto, como veremos, también lo podemos hacer con los condicionales).

Por último, las funciones son elementos esenciales en programación por dos motivos: permite reutilizar el código de nuestro programa no sólo en el mismo programa, si no en otros posteriores, y además, permite estructurar las aplicaciones de forma que podamos dividir el problema en otros más pequeños. De ahí que a la programación con procedimientos y funciones se la llame estructurada (y no funcional como suelen indicar algunos neófitos: la programación funcional, que existe, es totalmente diferente de la programación con funciones). Veremos cómo reunir código en porciones llamadas funciones y usarlo varias veces en otras partes del código de nuestros programas.

For

Un bucle **for** nos permite repetir un bloque de código según unas condiciones concretas, siempre que estas sean verdaderas. Un bucle *for* tiene tres partes:

- **inicialización**: en esta parte, debemos indicar las variables que vamos a usar en la condición. Lo más habitual es declarar variables numéricas, pero pueden ser perfectamente cadenas de caracteres, booleanos u objetos.
- **condición**: una o varias condiciones (unidas por operadores lógicos), que deciden el número de iteraciones del bucle.
- **operación**: una expresión que normalmente (aunque no es imprescindible) modificará las variables definidas en la primera parte.

Un ejemplo simple de bucle *for* sería:

```
for (x = 1; x < 10; x++)
{
  document.write("El número es " + x + "<br>");
}
```

Vamos a analizar este ejemplo. Fijémonos en la primera linea, donde podemos encontrar las tres partes que indicábamos antes:

- *x = 1* (inicialización): aquí usamos una variable *x* y le asignamos el valor 1, que será el valor de inicialización del bucle.
- *x < 10* (condición): nos encontramos con una condición muy simple, y que se leería como: "si *x* tiene un valor menor que 10, la condición es válida". Si la condición es válida, el bucle seguirá ejecutandose invariablemente. Veremos que eso puede generar un pequeño "problema", que a veces podemos incluso aprovechar para nuestros propósitos, el denominado "**bucle infinito**".
- *x++* (operación): aquí nos encontramos con una operación matemática usando un operador tipográfico "*++*", que viene a ser la misma operación que "*x = x + 1*". De hecho, también podemos usarla en vez de la que aparece, pero lo normal es que usemos un operador como éste (de hecho, si estudiamos código realizado por otros, veremos que es la forma más común).

Por tanto, el bucle *for* puede leerse como: "*Iniciamos 'x' con valor 1; el bucle se ejecutará mientras el valor de 'x' sea menor que 10, y cada vez que se ejecute el bucle el valor de 'x' se incrementará en 1*"

Modificación de la variable de control dentro del bucle

A continuación podemos ver un ejemplo de cómo podemos usar la variable de control del bucle dentro de éste:

```
for (x = 1; x < 16; x++)
{
   if (x % 2) {
      x++;
    }
    document.write('Mi número es ' + x + '<br>');
}
```

Podemos apreciar en este ejemplo que el resultado no es el aparentemente esperado para este bucle, dado que aunque en la parte de operación incrementamos la variable sólo en 1, el resultado final es que sólo vemos números pares en la ventana. ¿Por qué? Fijémonos en el código del bucle:

- el condicional if sólo será cierto en el caso de que el módulo de 'x' (la operación matemática de 'resto', representada por el operador '%') devuelva como resultado un valor impar
- si la condición de if se cumple, se ejecutará un trozo de código que incrementará el valor de 'x' en uno, influyendo de esta forma e el desarrollo del bucle.

De esta forma, en vez de mostrar los valores numéricos del 1 al 15, como sería en el caso normal, resulta que aparecen sólo valores pares. El truco en este código es sibilino pero potente: la operación 'x % 2' devolverá un 0 si 'x' es par, pero devolverá un valor distinto si es un número impar. ¿Esto que significa? Si el valor de la operación es cero, la condición es falsa (es un comportamiento muy habitual en muchos lenguajes de programación), con lo que la condición sólo será verdadera cuando 'x' sea impar. Un número impar devolverá un módulo mayor que cero al dividirlo entre 2. Al incrementar un valor impar en 1, se convierte en un valor par, porque es lo que finalmente vemos en pantalla.

Parece enrevesado, pero un estudio a fondo de este código nos permite apreciar algunas de las propiedades ocultas de la programación en JavaScript. Muchas de estas propiedades y características se aprenden con la experiencia, pero en este curso veremos algunas de ellas que nos serán útiles en nuestro trabajo futuro.

Bucles anidados

Podemos anidar varios bucles uno dentro de otro, como en este caso:

```
for (x = 1; x < 10; x++)
   for (y = 1; y < 10; y++)
      document.write(x + ":" + y);
```

En este ejemplo, vemos que un bucle se ejecutará dentro del otro mostrándonos los valores de forma ordenada. No es preciso escribir llaves si el código a escribir es de una sola linea.

Bucles infinitos

Vamos a ver rápidamente cómo realizar un bucle infinito con *for*:

```
for (;;)
{
 document.write("Esto no se acaba nunca...");
}
```

Esto genera un pequeño problema... Si este bucle se ejecuta constantemente... ¿Qué hacemos para detenerlo o controlarlo? Se pueden utilizar dos palabras para controlar un bucle (sea cual sea), ya sea finito o infinito: **break** y **continue**.

Sin embargo, **LAS BUENAS PRÁCTICAS DE PROGRAMACIÓN DESACONSEJAN TOTALMENTE EL USO EN CUALQUIER LENGUAJE DE PROGRAMACIÓN DE LAS SENTENCIAS *break* Y *continue* COMO MECANISMO DE CONTROL DEL FLUJO DEL PROGRAMA**

`break` es la palabra reservada para cortar un bucle en un momento determinado. Es muy usada en condicionales if, ya que al darse una cierta condición, podemos controlar un bucle cortándolo cuando se dé un cierto momento concreto.

`continue`, sin embargo, también corta la ejecución del bucle, pero no igual que break. Mientras que break finaliza definitivamente el bucle, continue salta lo que queda de bucle y sigue la siguiente iteración (repetición) sin más.

```
var x = 1;
for (;;)
{
```

```
   x++;
   if (x > 5) break;
   document.write(x + '<br>');
}
```

En este ejemplo vemos que, cuando el valor de *x* sea mayor de 5, el bucle se romperá. En este otro nos permite apreciar el resultado de usar *continue*:

```
for (x = 1; x < 10; x++)
{
   if (x % 2) continue;
   document.write(x + '<br>');
}
```

Ya nos podemos imaginar el resultado.

For sobre listas

Un caso particularmente útil es la utilización de bucles for para recorrer los elementos de una lista (o array). Podemos utilizar la sintaxis anterior recorriendo el array como en el siguiente ejemplo:

```
var lista = ["elemento1","elemento2","elemento3"];
for (idx=0; idx<lista.length; idx++)
{
   elemento_n = lista[idx];
   alert(elemento_n);
}
```

Sin embargo dado que su uso es tan habitual existe una sintaxis abreviada que veremos frecuentemente al consultar código javascript en Internet:

```
var lista = ["elemento1","elemento2","elemento3"];
for (elemento_n in lista) {
   alert(elemento_n);
}
```

elemento_n tomará sucesivamente todos los valores de la lista (array) sin necesidad de utilizar un índice (*idx*), haciendo el código más compacto.

While

El bucle *while*, al igual que el bucle *for*, también es condicional, aunque mucho más simple que éste, como vemos en el siguiente ejemplo:

```
var x = 1;
while (x < 10)
{
   document.write("Mi número es: " + x + "<br>");
   x++;
}
```

Como vemos, este bucle es idéntico al bucle *for* que estudiamos en el apartado anterior pero mucho más simple.

¿Para qué hay dos bucles que al fin y al cabo hacen los mismo? En el pasado *for* era un bucle incondicional, es decir, que sólo podía ir de cierto punto a cierto punto, sin comprobaciones intermedias. Por eso se desarrolló el bucle *while*. Sin embargo, lenguajes más avanzados como C crearon un *for* condicional más potente. Aunque realiza funciones semejantes, *while* tiene como ventaja que, al ser más simple, también es más rápido y eficaz en términos de rendimiento que *for*. Además, algunas operaciones resultan simplificadas:

```
while (verdad)
{
   if (verdad) document.write("Es verdad");
   verdad = !verdad;
}
```

Este extraño ejemplo es significativo por el uso de variables lógicas o booleanas. *verdad* es una variable lógica que sólo puede tener los valores *true* y *false*. Si *verdad* es *true*, el bucle se ejecutará. Al ejecutarse, también se comprobará la condicion interna y se imprimirá el mensaje "Es verdad". Lo interesante viene después. La expresión *verdad = !verdad* significa "hacer que *verdad* sea contrario a *verdad*". Si *verdad* era igual a *true*, ahora será igual a *false* (el operador *!*, como sabemos, es la negación, y por lo tanto, cambiará el valor de la variable a su contrario).

while tiene una versión inversa: *do-while*. Este bucle tiene como diferencia respecto de *while* que, mientras que en *while* la condición se comprueba incluso antes de comenzar a ejecutar el bucle (lo que implica que si la condición ya fuese falsa antes de entrar en el bucle, éste no llegaría a ejecutarse nunca), en *do-while* la condición se comprueba a posteriori, con lo que tenemos la oportunidad de ejecutar el bucle al menos una vez. Esto también es

una ventaja con respecto a *for*, que en ese sentido se comporta igual que *while*. Veamos un ejemplo:

```
var x = 0;

do
{
   document.write("Mi número es el " + x + "<br>");
   x++;
} while (x < 10);
```

Funciones

¿Qué son las funciones?

Las funciones son uno de los elementos más importantes de cualquier lenguaje de programación actual. De hecho, Niklaus Wirth, uno de los más importantes teóricos de la programación y creador del lenguaje Pascal entre otros, llegó a indicar que todo programa no era más que la suma de código (rutinas, procedimientos o funciones, como se les quiera llamar) y datos (variables, matrices, etc...). Sea como sea, las funciones tienen un papel estelar en el desarrollo de aplicaciones en la actualidad.

Hasta ahora, hemos visto como realizar código de una forma estructurada, con sentencias de control que nos permiten dominar la ejecución del mism fácilmente. Pero si sólo tenemos esto, solamente podremos manejar el código de forma lineal: ejecutaremos las líneas una a una, una detrás de la otra, hasta el final del programa. Es más, si quisiéramos usar un determinado código varias veces en el mismo programa tendríamos que repetir ese código varias veces, teniéndolo que adaptar a cada situación. ¿Y qué ocurre si queremos reutilizar un código en varios programas? Es un problema que se resuelven gracias a las funciones.

Las funciones son trozos de código que tienen un nombre y que podemos utilizar en cualquier parte de nuestro código con una llamada directa. Este es un buen ejemplo:

```
function datos_personales(nombre, apellidos, edad)
{
   return "Hola, " + nombre + " " + apellidos + ", tienes " + edad + " años.";
}
```

En este caso hemos definido una función que, usando los parámetros que le hemos pasado, los combina para formar una cadena formateada, que devuelve gracias a la palabra reservada *return*. ¿Y cómo podemos usar este código?

```
alert(datos_personales('Pepito', 'Pérez', 25));
```

De tal forma que primero ejecutaremos la función *datos_personales* con los parámetros pasados, y después la función alert, que nos permite mostrar una ventana con un mensaje en la pantalla, con el resultado devuelto por la función que hemos creado. Este sería el código completo del programa:

```
<html>
<head>
   <title>código de función</title>
   <script>
   function datos_personales(nombre, apellidos, edad)
   {
      return 'Hola, ' + nombre + ' ' + apellidos + ', tienes ' + edad + ' años.';
   }
   </script>
</head>
<body>
   <script>
   alert(datos_personales("Pepito", "Perez", 25));
   </script>
</body>
</html>
```

Los parámetros son un elemento importante en las funciones. Se trata de datos que podemos pasar a nuestras funciones para que estas los procesen de alguna forma, lo cual dará como resultado o una salida en el navegador (texto o ventanas), o bien un resultado que se puede usar cuando llamemos a la función. Cuando indiquemos parámetros, deberemos indicar la lista completa de parámetros que vamos a usar en esa función. Al llamar a la función, deberemos hacerlo con todos los parámetros indicados. De otra forma, se produciría un error de ejecución. Los parámetros se convierten en variables de datos dentro de la función, de ahí que podamos utilizarlas e incluso modificarlas.

Generalmente, las funciones se utilizan para realizar alguna operación no visible (matemática, de cadena de caracteres, de objetos, etc...) que devuelv por medio de *return*, pero también se pueden visualizar elementos en el navegador usando las funciones y objetos que ya incorpora JavaScript.

```
function suma(dato1, dato2)
{
   return dato1 + dato2;
}
```

Al utilizar esta función, podemos hacerlo de varias formas:

```
var total = suma(1,2);   // 3
alert(suma(7,43));   // 50
document.write(total + suma(54,-7));  // 50
```

A lo largo de los siguientes capítulos veremos como crear y utilizar funciones, tanto propias como ajenas. Recordemos que podemos incluir código desde otro archivo y por supuesto, ese código puede contener funciones. En el siguiente capítulo veremos más ejemplos de funciones y avanzaremos en funciones internas de JavaScript.

Funciones con parámetros

Como ya indicamos en el capítulo anterior, los parámetros nos sirven para llamar a nuestras funciones con unos datos específicos para que los procese. Y en cada llamada, podemos darle unos parámetros diferentes, que harán que pueda comportarse de forma diferente, si ese es el comportamiento que le hemos programado.

```
var numero = 1;
var cadena = "Hi!";
var logico = true;

function valores(num, cad, log)
{
   document.write(num);
   document.write(cad);
   document.write(log);
}

valores(numero, cadena, logico);
```

Esta función la estamos llamando con variables como parámetros, pero también podemos llamar a la función con valores literales, es decir, valores simples directos:

```
valores(2, "adiós", false);
```

Como ya vimos en el capítulo anterior, también podemos hacer que otra función sea un parámetro:

```
valores(3, "que tal".length, true);
```

"que tal".length es una función que forma parte de los objetos de cadena de texto (todo, desde las variables hasta los literales, son objetos en JavaScript), y nos devuelve la longitud de una cadena de texto. En concreto, al hacer esta llamada nos devolverá un número '7'. Como las variables en JavaScript no tienen tipo (todas son objetos), podemos pasar cualquier valor como parámetro.

Devolución de datos

Como ya sabemos, una función puede devolver datos hacia afuera por medio de la expresión **return**. Naturalmente, podemos devolver cualquier tipo de datos. Sin embargo hay que tener en cuenta una serie de cuestiones:

- Siempre se devuelven objetos, como ya hemos visto, y por lo tanto podemos devolver un objeto creado en la misma función. Normalmente, cuando creamos una variable dentro de una función, esta variable existe sólo para esa función, y desaparece en el momento en que la función termina (la variable se encuentra en la pila de memoria, y cuando la función desaparece, también lo hace la pila); pero en el caso de que devolvamos el objeto, no se devuelve exactamente la misma variable, si no que se devuelve su contenido.
- Cuando devolvemos *true* ó un valor distinto que cero, para JavaScript es lo mismo, y si devolvemos *false* o 0, también viene a ser lo mismo. Esta es una regla estándar para muchos lenguajes como JavaScript, Java, PHP, Perl, etc...
- No es preciso que una función devuelva nada. No es necesario usar *return*. Además, también es posible que en vez de devolver resultados, se modifiquen variables globales, es decir, variables creadas fuera de la función y que se usan dentro.
- Si queremos salir de una función antes de tiempo, porque algo ha fallado o no hay nada que hacer en un caso específico, podemos simplemente escribir "return;", lo que nos permitirá salir sin más y no devolver ningún valor

Estas consideraciones son importantes a nivel general y es importante tenerlas en cuenta. Vamos a ver como funcionan con algunos ejemplos:

```
function dev_variable()
{
   variable = true;
   return variable;
}

var var1 = dev_variable();
```

Como vemos, hemos declarado una variable local a la función y la hemos devuelto, pero solo se devuelve realmente el valor. Esto pasa en todos los casos (Nota técnica: cuando se devuelve un objeto, se devuelven sus datos en forma de objeto de esa clase; esto lo entenderemos mejor en el capítulo siguiente). Veamos este otro ejemplo:

```
function dev_true() {
   return true;
}

if (dev_true()) {
   alert("es true");
}

if (true) {
   alert("también es true");
}

if (1)
{
   alert("este también es true");
}
```

Por último, veamos cómo salir de una función sin necesidad de devolver nada en cualquier momento:

```
function salir()
{
   document.write("hola");
   document.write("que pasa");
   return;
   alert("adiós");
}

salir();
```

En este ejemplo, la última linea dentro de la función (*alert*) no se ejecutará nunca porque hemos salido sin más en la linea anterior al ejecutarse la instrucción *return*.

Funciones recursivas

Las funciones recursivas son aquellas que se llaman a sí mismas. Existen multitud de técnicas para desarrollar este tipo de funciones, ya que sus usos son muy diversos, pero fundamentalmente hay que tener en consideración que son funciones peligrosas, porque si no controlamos su ejecución, se estarán ejecutando indefinidamente, como en el caso de los bucles infinitos. La diferencia con los bucles infinitos es que dependiendo de la implementación del intérprete de JavaScript, es posible que rompamos la pila de memoria, que ya vimos antes, con lo que además de colgar el navegador, podemos generar una excepción de memoria y un error grave del sistema. Para evitarlo, claro está, debemos estudiar bien la lógica de la función para construirla adecuadamente. Por ejemplo, si queremos calcular el factorial de un número, podemos hacerlo con una función recursiva:

```
function factorial(numero)
{
   if (numero == 1 || numero == 0)
      return 1;
   else
      return numero*factorial(numero - 1);
}

document.write(factorial(4));
```

Supóngase la llamada a esta función para N=4, es decir *factorial(4).* Cuando se llame por primera vez a la función, la variable *numero* valdrá 4, y por tanto devolverá el valor de 4*factorial(3); pero factorial(3) devolverá 3*factorial(2); factorial(2) a su vez es 2*factorial(1) y dado que factorial(1) es igual a 1 (es importante considerar que sin éste u otro caso particular, llamado caso base, la función recursiva no terminaría nunca de llamarse a sí misma), el resultado final será 4*(3*(2*1)).

Funciones en JavaScript

JavaScript contiene sus propias funciones que podemos utilizar en nuestros programas. Aunque algunas de estas funciones podemos usarlas independientemente de sus correspondientes objetos, lo cierto es que todas las funciones provienen de algún objeto específico. El objeto **window** representa a la ventana del navegador y es el objeto por defecto. Esto quiere decir que podemos usar sus elementos (funciones, propiedades, objetos, etc...) sin necesidad de llamar explícitamente al objeto *window*.

Ya conocemos alguna que otra función como *length* (de los objetos de matriz) o *alert*, que proviene del objeto *window* y que muestra un mensaje en una ventana.

Otro objeto dependiente de window es **document** que contiene, entre otras cosas, funciones como **write** que nos permite escribir texto en la página web.

A continuación vamos a estudiar algunas posibilidades que nos aportan las funciones en JavaScript, ya que hay una gran cantidad de ellas. Sin embargo, vamos a repasar las más usadas para el desarrollo web en general.

Mostrar ventana de confirmación (aceptar o cancelar)

```
<html>
<head>
<script type="text/javascript">
   function ver_confirm()
   {
      var name=confirm("Pulsa un botón")
      if (name==true) {
         document.write("Has pulsado el botón Aceptar");
      }
      else {
         document.write("Has pulsado el botón Cancelar");
      }
   }
</script>
</head>

<body>
<form>
   <input type="button" onclick="ver_confirm()" value="Mostrar ventana confirmación">
</form>
</body>
</html>
```

Abrir una ventana nueva

```
<html>
<head>
<script language="javascript">
   function open_win()
   {
      window.open(" http://www.google.es ","nueva","toolbar=yes, location=yes,
                  directories=no, status=no, menubar=yes, scrollbars=yes,
                  resizable=no, copyhistory=yes, width=400, height=400");
   }
</script>
</head>

<body>
<form>
   <input type="button" value="Abrir ventana" onclick="open_win()">
</form>
</body>
</html>
```

El primer parámetro de **open** es la dirección que queremos mostrar en la ventana. El segundo es el nombre que queremos darle a la ventana (y que podemos usar, por ejemplo, en el atributo *target* de los enlaces). El tercer y último parámetro nos permite definir el aspecto de la ventana según los datos que le indicamos.

Cambiar la URL actual

El siguiente ejemplo nos muestra el uso de **location** para conseguir la dirección de la página actual en la que nos encontramos o bien ir a una página diferente:

```
<html>
<head>
<script type="text/javascript">
   function actual_location()
   {
```

```
      alert(location);
   }
   function cambiar_location()
   {
      window.location=" http://www.google.es/ ";
   }
</script>
</head>

<body>
<form>
   <input type="button" onclick="actual_location()" value="Mostrar la URL actual">
   <input type="button" onclick="cambiar_location()" value="Cambiar URL">
</form>
</body>
</html>
```

Imprimir una página

```
<html>
<head>
<script type="text/javascript">
   function printpage()
   {
      window.print();
   }
</script>
</head>

<body>
<form>
   <input type="button" value="Imprime esta página" onclick="printpage()">
</form>
</body>
</html>
```

Cambiar el tamaño de la ventana actual

```
<html>
<head>
<script type="text/javascript">
   function resizeWindow()
   {
      window.resizeBy(-100,-100)
   }
</script>
</head>

<body>
<form>
   <input type="button" onclick="resizeWindow()" value="Redimensionar ventana">
</form>
</body>
</html>
```

Si se usan frames, debe utilizarse el elemento **top** en vez del elemento **window**, para representar el frame superior

Avanzar a una posición específica de la ventana

```
<html>
<head>
<script type="text/javascript">
   function scrollWindow()
   {
      window.scrollTo(100,500)
   }
</script>
</head>
<body>
<form>
   <input type="button" onclick="scrollWindow()" value="Scroll">
</form>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
<br><br><br><br><br><br><br><br>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
<br><br><br><br><br><br><br><br>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
<br><br><br><br><br><br><br><br>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
<br><br><br><br><br><br><br><br>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
<br><br><br><br><br><br><br><br>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
```

```
<br><br><br><br><br><br><br><br>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
<br><br><br><br><br><br><br><br>
<p>SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL SCROLL</p>
<br><br><br><br><br><br><br><br>
</body>
</html>
```

Retrasar la ejecución del programa durante un tiempo determinado

```
<html>
<head>
<script language="javascript">
   function timeout()
   {
      setTimeout("alert('Esta ventana aparece un segundo después de que
                  hayas pulsado el botón')", 1000)
   }
</script>
</head>

<body>
<form>
   <input type="button" onclick="timeout()" value="Cuenta atrás">
</form>
</body>
</html>
```

Un par de detalles sobre este último ejemplo: La función *setTimeout* tiene dos parámetros: una cadena de texto que representa un código JavaScript a ejecutar cuando hayan pasado el número de milisegundos del segundo parámetro.

Cambiar una imagen por otra

```
<html>
<head>
<script type="text/javascript">
   function setSrc()
   {
      var x=document.images
      x[0].src="foto1.gif"
   }
</script>
</head>
<body>
<img src="foto2.gif" width="107" height="98">
<form>
   <input type="button" onclick="setSrc()" value="Cambiar imagen">
</form>
</body>
</html>
```

Como podemos apreciar, lo primero que hacemos es recoger todas las imagenes de la página en una matriz mediante el objeto **document.images**y después acceder a la imagen específica como se muestra en la funcion *setSrc().* Para hacer el cambio usamos la propiedad **src** que tienen todos los objetos de imagen.

A medida que vayamos pasando por los capítulos de este curso, iremos viendo nuevos ejemplos con nuevas posibilidades de manejo de las funciones y sus respectivos objetos. En el siguiente capítulo veremos cómo crear nuestros propios objetos y aplicarlos en nuestros programas.

Clases y objetos

Dentro de los lenguajes actuales, que provienen en su mayoría de los primeros lenguajes estructurados como ALGOL o BCPL, la capacidad de crear funciones para reutilizarlas de diversas formas, ya sea dentro del mismo módulo del programa o en distintos módulos, ha sido uno de los fundamentos de la programación de sistemas informáticos. Sin embargo, este paradigma se quedaba corto para nuevas situaciones que iban surgiendo con el tiempo, como la programación de videojuegos o el 3D y la inteligencia artificial. A finales de los años 70 se comenzó a teorizar sobre lenguajes de programación que utilizasen entidades independientes que fueran autocontenidas para realizar las aplicaciones. Como ya dijimos anteriormente, un programa se compone de código y datos. los objetos son unidades que contienen su propio código y datos para su propio auto-funcionamiento. Podemos decir que son programas dentro de programas.

Dicho así, podemos darnos cuenta de que los objetos pueden ser utilizados como variables, para nuestro propio uso. Pero no podemos definir variables de objeto sin poder darles una forma. La forma de los objetos son los datos (propiedades) y código (funciones) que contiene el objeto. A esi se le denomina clase de objeto. Para definir clases en JavaScript, lo hacemos por medio de funciones, como esta:

```
function Persona(nombre) {
        this.nombre = nombre;
        this.color_pelo = 'negro';
        this.peso = 75;
        this.altura = 165;
        this.sexo = 'varón';
        this.edad= 26;
}
```

Vamos a fijarnos bien como se estructura esta función. Se le llama constructor de la clase, y en ella definimos los datos de la clase, los que vamos a poder utilizar al crear objetos con ella. Nótese el uso de la palabra reservada this. Esta palabra sirve para identificar al objeto en sí mismo dentro de la definición de la clase. Cuando escribimos

```
this.peso = 75;
```

estamos creando la propiedad "peso" de la clase "Persona". Cuando creamos una propiedad en el constructor y le damos un valor, como en este caso, estamos asignándole un valor por defecto. Todos los objetos creados con este constructor contendrán una propiedad "peso" con ese valor inicial, aunque luego podremos cambiarlo al usar el objeto. Para definir un objeto de esta clase, sólo tendríamos que hacer esto:

```
var hombre = new Persona('Pepe');
```

Aquí hemos definido el objeto "hombre", que contendrá todas las propiedades definidas en la función-clase "Persona". Si queremos cambiar su valor, sólo tenemos que hacer algo como esto:

```
hombre.peso = 80;
```

De esta forma, el dato definido para este objeto cambia. Pero si hemos definido más objetos de tipo Persona, cada uno de ellos contendrá las mismas propiedades pero con valores distintos. Ningún objeto tiene el mismo valor que otro objeto de la misma clase a no ser que nosotros se lo asignemos explícitamente.

```
var mujer = new Persona('Ana');
mujer.peso = 67;
mujer.sexo = 'mujer';
```

En este caso hemos hecho lo mismo, pero le indicamos su propio peso, independiente del de la variable "hombre". Así, podemos tener tantos objetos de la misma clase como queramos para realizar las operaciones que sean pertinentes. Una última cosa sobre los constructores: como podemos ver, podemos pasarle parámetros, que podemos convertir en los valores de las propiedades de los objetos de esa clase.

Creación de funciones miembro

Hasta ahora hemos visto como crear propiedades de las clases, pero necesitamos crear código en ese objeto que utilice las propiedades que hemos creado en el constructor Para crear una función miembro, debemos indicarlo en la propia función de construcción:

```
function Persona(nombre) {
        this.nombre = nombre;
        this.color_pelo = 'negro';
        this.peso = 75;
        this.altura = 165;
        this.sexo = 'varón';
        this.dormir = dormir; // Nueva función miembro
}
```

Y ahora definimos la función dormir:

```
function dormir() {
        alert(this.nombre + ' está durmiendo' );
}
```

Fijémonos en la función. Tiene una forma bastante normal. Lo único especial que hemos hecho es añadir la línea

```
    this.dormir = dormir;
```

al constructor, con lo que hemos asignado la función dormir como si fuera una propiedad. Recordemos que TODO es un objeto en JavaScript, y esto incluye a las funciones. Ahora, para ejecutar este código, utilizamos el objeto anteriormente creado para ponerlo en marcha:

```
    hombre.dormir();
```

Veamos en un ejemplo todo el código que hemos generado hasta ahora:

```
    <html>
    <head>
    <script language="javascript">
    function Persona(nombre) {
        this.nombre = nombre;
        this.color_pelo = 'negro';
        this.peso = 75;
        this.altura = 165;
        this.sexo = 'varón';
        this.dormir = dormir;
    }

    function dormir() {
            alert(this.nombre + ' está durmiendo' );
    }
    </script>
    </head>

    <body>
    <form>
    </form>
    <script>
        var hombre = new Persona('Pepe');
        hombre.dormir();
    </script>
    </body>
    </html>
```

Como resultado, nos mostrará el mensaje "Pepe está durmiendo". Como vemos, podemos usar las propiedades de los objetos dentro de las funciones miembro, aunque también podríamos construir la misma función de otra manera:

```
    function dormir() {
        with (this)
            alert(nombre + ' está durmiendo' );
    }
```

`with` es una palabra reservada de JavaScript que permite coger una variable de objeto como this y permite utilizar sus miembros como si fueran variables independientes. Pero tiene sus restricciones: estos nombres abreviados sólo se pueden utilizar dentro del ámbito de with (que si tiene varias líneas, estas deben estar contenidas entre llaves, como for, if, etc...), y además, se pueden confundir fácilmente con variables locales a la función o globales del programa, con lo cual particularmente no recomendamos el uso de with, ya que puede dar lugar a fallos de ejecución difíciles de tratar si no se tienen en cuenta estas restricciones. Se aconseja usar la forma this.nombre. También se recomienda crear cada clase en un archivo diferente para que no haya confusiones de nombres, sobre todo de funciones miembro.

Otra manera de declarar la clase en JavaScript:

```
 <html>
    <head>
    <script language="javascript">
    function Persona(nombre) {
        this.nombre = nombre;
        this.color_pelo = 'negro';
        this.peso = 75;
        this.altura = 165;
        this.sexo = 'varón';
        this.dormir = function dormir(){
            alert(this.nombre + ' está durmiendo' );
        }

    }
    </script>
    </head>
    <body>
    <form>
    </form>
    <script>
        var hombre = new Persona('Pepe');
        hombre.dormir();
    </script>
    </body>
    </html>
```

Con este ejemplo se obtiene el mismo resultado que el anterior pero el código queda un poco mas complejo. A pesar de esto ya podemos ver que a diferencia del código anterior este se encuentra encapsulado en la misma función *[function Persona(){}]*

Otro ejemplo de creación de una clase más complicado **utilizando DOM estándar y JavaScript**; *debemos recordar que innerHTML es un método propietario de Microsoft y que desaparecerá de la especificación en un futuro muy próximo.* Nótese que la propiedad texto debe ser un nodo de texto (DOMTextNode) en lugar de HTML (Cualquier interface DOM: DOMNode, DOMElement...) al uso como ocurriría usando innerHTML.

```
<html>
    <head>
    <script type="text/javascript"  charset="utf-8">
```

```
    // <![CDATA[
        function CrearCapas(idcapa, info) {
            this.id          = idcapa;
            this.texto       = info;
            this.CrearCapa   = function CrearCapa() {
                try {
                    // Esta es la manera correcta y estándar -aunque más lenta y costosa-
                    // de generar contenido mediante el DOM:
                    // Objetos DOMElement necesarios:
                    var body    = document.getElementsByTagName('body').item(0);    // Tag <body> también se puede en
forma de vector:
                                                                    //...agName('body')[0]; pero no es compatible
con Opera.
                    var capa    = document.createElement('div');                    // División al vuelo
                    var texto   = document.createTextNode(this.texto);          // Texto contenido en div al vuelo
                    // Establecer atributos de división:
                    capa.setAttribute('id', this.id);
                    capa.setAttribute('style', 'background-color:#f7f7f7; width:100px; border:#000000 2px solid;'  );
                    // Reconstruir el árbol DOM:
                    capa.appendChild(texto);
                    body.appendChild(capa);
                } catch(e) {
                    alert(e.name + " - " + e.message);
                }
            }
        }
    // ]]>
```

```
    </script>
    </head>

    <body>
        <script type="text/javascript"  charset="utf-8">
```

```
        // <![CDATA[
            var capa = new CrearCapas('idCapanueva', 'Hola Mundo');
            capa.CrearCapa();
        // ]]>
```

```
        </script>
    </body>
    </html>
```

El resultado del código anterior es una página que por medio de JavaScript crea una división (div) y le asigna atributos como ancho, alto, color, etc, y lo inserta dentro de la página al cargarse.

Creando clases usando el estándar ECMAScript 6

Con la llegada del stándar ECMAScript 6 en el 2015 se agregó azúcar sintáctica a la forma de crear las clases en JavaScript, ahora podemos crear nuestras clases de una forma más sencilla y estandarizada como lo hacen los lenguajes de POO.

Creando una clase

```
1 // Para crear una clase se usa la palabra reservada 'class'
2 class Persona {
3     /* El método 'constructor' se utiliza para inicializar los atributos
4      * de la clase.
5      *
6      * Observar que se pueden especificar valores por defecto a los perámetros.
7      *  - Se pasa el valor por defecto 'conocido' a 'tipoSaludo'.
8      */
```

```
 9      constructor(nombre, edad, email, tipoSaludo = 'conocido')
10      {
11          // Para hacer referencia a las propiedades del objeto se utiliza la
12          // palabra reservada 'this'.
13          this._nombre = nombre;
14          this._edad = edad;
15          this._email = email;
16          this._tipoSaludo = tipoSaludo;
17      }
18
19      // Se pueden crear los getter con la palabra reservada 'get'.
20      // Los getter sirven para obtener los valores de las propiedades
21      // del objeto.
22      get nombre()
23      {
24          return this._nombre;
25      }
26
27      // Se pueden crear los setter con la palabra reservada 'set'.
28      // Los setter sirven para asignar nuevos valores a las propiedades
29      // del objeto.
30      set tipoSaludo(tipoSaludo) {
31          this._tipoSaludo = tipoSaludo;
32      }
33
34      // Para crear un método simplemente se define su nombre
35      saludar(nombre)
36      {
37          if (this._tipoSaludo === 'conocido')
38              console.log(`Hola ${nombre}, ¿Cómo estas?`);
39          else
40              console.log(`Hola, mi nombre es ${this._nombre}`);
41      }
42
43      /* En algunas ocaciones se puede dar el caso de que no podemos tener
44       * acceso a nuestro objeto, la solución a este inconveniente se muestra
45       * y explica en este método.
46      */
47      mostrarme()
48      {
49          // Declarando una variable local y asignándole una referencia al propio
50          // objeto.
51          let _this = this;
52
53          // En una función anónima no se puede acceder al propio objeto usando
54          // la palabra reservada 'this' (obtenemos como salida 'undefined').
55          (function () {
56              console.log(this);
57          })();
58
59          // Una solución es declarar una variable y asignarle una referencia
60          // al objeto como se hace al inicio del método.
61          (function () {
62              console.log(_this);
63          })();
64
65          // Esta es la manera correcta y elegante de acceder a nuestro objeto.
66          ((e) => {
67              console.log(this);
68          })();
69      }
70
71      // Los métodos estáticos se declaran usando la palabra reservada 'static'.
72      static girar()
73      {
74          console.log('Girando!');
75      }
76  }
77
78  // Para crear una instancia de la clase 'Persona' se usa la palabra reservada
79  // 'new'. Recordar que el cuarto parámetro es opcional, por lo que al no pasarle
80  // valor tomara por defecto el especificado en el método 'constructor' de la
81  // clase.
82  var p = new Persona('Juan', 32, 'juan@mail.com');
83
84  // Llamando a uno de sus métodos.
85  p.saludar('Ana');
86
87  // Cambiando el valor del atributo 'tipoSaludo' usando el setter tipoSaludo
88  p.tipoSaludo = 'otro';
89  p.saludar();
90
91  // Obtenieno el valor del atributo 'nombre' usando el getter nombre
92  console.log(p.nombre);
93
94  // Ejemplo del acceso al propio objeto y la mejor forma de hacerlo, en
95  // circunstancias como: los eventos, funciones anónimas, uso de JQuery dentro
96  // del método, etc.
97  p.mostrarme();
98
99  // Un método estático no necesita de una instacia de clase para ser invocado.
100 Persona.girar();
```

Clases predeterminadas en JavaScript

A continuación vamos a estudiar algunos de los objetos y clases más utilizados en JavaScript.

Clase Array

La clase **Array** permite crear una matriz de datos. Vamos a estudiar algunos de sus métodos con el siguiente ejemplo:

```
<html>
<body>
<script type="text/javascript">
   var famname = new Array(3);
   famname[0] = "Jani";
   famname[1] = "Tove";
   famname[2] = "Hege";

   document.write(famname.length + "<br/>");
   document.write(famname.join(".") + "<br/>");
   document.write(famname.reverse() + "<br/>");
   document.write(famname.push("Ola","Jon") + "<br/>");
   document.write(famname.pop() + "<br/>");
   document.write(famname.shift() + "<br/>");
</script>
</body>
</html>
```

Estudiemos el código. Después de crear la matriz, utilizamos algunas funciones y propiedades.

- **length** sirve para conocer la cantidad de elementos que contiene la matriz propiamente dicha.
- **join** permite unir todos los elementos separados por una cádena de caracteres que pasamos como parámetro, en este caso, ".".
- **reverse** posiciona los elementos actuales de forma inversa.
- **push** nos permite añadir un nuevo elemento dentro de la matriz (en realidad, podemos añadir cualquier cantidad de ellos).
- **pop** extrae el último elemento de la matriz y lo devuelve.
- **shift** extrae y devuelve el primer elemento de la lista.

Si queremos ordenar los elementos de una matriz, podemos usar la función miembro **sort** para realizar esta operación. Pero la ordenación realizada es "lexicográfica", es decir, que se ordenarán alfabéticamente. Si queremos realizar una ordenación númerica, podemos crear una función de comparación como veremos en el siguiente ejemplo:

```
<html>
<body>
<p>
   Nota: Si no usamos función de comparación para definir el orden, la matriz
   se ordenará siempre alfabéticamente. "500" vendrá antes que "7", pero en una
   ordenación numérica, 7 viene antes que 500. Este ejemplo muestra como usar la
   funcion de comparación - que ordenará correctamente los elementos tanto en una
   matriz numérica como de cadenas.
</p>

<script type="text/javascript">
   array1 = new Array("Rojo","Verde","Azul");
   array2 = new Array("70","9","800");
   array3 = new Array(50,10,2,300);
   array4 = new Array("70","8","850",30,10,5,400);

   function compareNum (a, b)
   {
      return a-b;
   }

   document.write("Ordenado: " + array1.sort());
   document.write("<br><br>");

   document.write("Ordenado sin compareNum: " + array2.sort());
   document.write("<br>");
   document.write("Ordenado con compareNum: " + array2.sort(compareNum));
   document.write("<br><br>");

   document.write("Ordenado sin compareNum: " + array3.sort());
   document.write("<br>");
   document.write("Ordenado con compareNum: " + array3.sort(compareNum));
   document.write("<br><br>");

   document.write("Ordenado sin compareNum: " + array4.sort());
   document.write("<br>");
   document.write("Ordenado con compareNum: " + array4.sort(compareNum));
</script>
</body>
</html>
```

Como podemos apreciar en el código, sólo las llamadas a *sort* que tienen como parámetro la función de comparación *compareNum* han sido ordenadas numéricamente.

Clase Date

Esta clase permite definir una fecha y hora. Tiene una buena cantidad de funciones y aquí vamos a estudiar algunas de las más interesantes.

```
<html>
<body>
<script type="text/javascript">
   var d = new Date();
   document.write(d.getDate());
   document.write(".");
   document.write(d.getMonth() + 1);
   document.write(".");
   document.write(d.getFullYear());
</script>
</body>
</html>
```

Este ejemplo construye la fecha actual mediante los métodos **getDate**, **getMonth** y **getFullYear**. El valor base de *getMonth* es 0 (Enero). En los siguientes ejemplos veremos como adaptar el objeto a la fecha que nosotros queremos.

En el siguiente ejemplo extraeremos la hora actual:

```
<html>
<body>
<script type="text/javascript">
   var d = new Date();
   document.write(d.getHours());
   document.write(".");
   document.write(d.getMinutes());
   document.write(".");
   document.write(d.getSeconds());
</script>
</body>
</html>
```

La dinámica de este ejemplo es muy parecida al anterior, pero en este caso usamos **getHours**, **getMinutes** y **getSeconds**.

```
<html>
<body>
<script type="text/javascript">
   var d = new Date();
   d.setFullYear("1990");
   document.write(d);
</script>
</body>
</html>
```

Este ejemplo muestra cómo modificar el año, con **setFullYear**, aunque también podemos cambiar otras partes de la fecha y la hora, con **setMonth**, **setDate** (para el día), **setHours**, **setMinutes** y **setSeconds**. En vez de *setFullYear*, que tiene como parámetro un año con todas sus cifras, podemos usar también **setYear**, que sólo necesita las dos últimas cifras del año (de 00 a 99). Algo a tener en cuenta es que con esto no cambiamos ningún parámetro de la fecha y hora del sistema, si no del objeto Date exclusivamente.

En el siguiente ejemplo veremos como mostrar los días de la semana:

```
<html>
<body>
<script language="javascript">
   var d = new Date();
   var weekday = new Array("Domingo", "Lunes", "Martes", "Miercoles",
                           "Jueves", "Viernes", "Sábado");
   document.write("Hoy es " + weekday[d.getDay()]);
</script>
</body>
</html>
```

Como vemos, podemos crear una matriz con los días de la semana (comenzando por el domingo, dado que se toma la referencia anglosajona), y referenciar a sus elementos con la función *getDay*. También podemos usar otras funciones como *getMonth*, *getYear*, *getHours*, *getMinutes* y *getSeconds*.

Clase Math

Esta clase contiene funciones y propiedades relacionadas con las matemáticas.

```
<html>
<body>
<script language="javascript">
   document.write(Math.round(7.25) + "<br>");
   document.write(Math.random() + "<br>");
   no = Math.random()*10;
   document.write(Math.round(no) + "<br>");
   document.write(Math.max(2,4) + "<br>");
   document.write(Math.min(2,4) + "<br>");
</script>
</body>
   </html>
```

La función **round** permite redondear una cifra de coma flotante a un entero. **random** genera un número aleatorio, o si queremos que este número se encuentre entre 1 y 10, lo podemos hacer como en la siguiente linea, generando un número aleatorio y multiplicándolo por el máximo que queremos. **max** y **min** devuelven el número máximo y mínimo entre dos dados, respectivamente. A su vez, esta clase contiene también funciones trigonométricas como **cos**, **sin**, **tan**, **acos**, **asin**, **atan**. Podemos contar con otras funciones de coma flotante como **ceil**, **log**, y **sqrt** (raiz cuadrada). Como puede comprobarse también, no hace falta crear un objeto para usar esta clase (se las denomina clases estáticas).

Clase String

Esta clase permite la manipulación de cadenas de texto. Toda cadena de texto que creamos es un objeto de esta clase, así que podemos hacer manipulaciones de muy diverso tipo.

```
<html>
<body>
<script type="text/javascript">
   var str = "¡JavaScript es malo!";

   document.write("<p>" + str + "</p>");
   document.write(str.length + "<br>");

   document.write("<p>" + str.fontcolor() + "</p>");
   document.write("<p>" + str.fontcolor('red') + "</p>");
   document.write("<p>" + str.fontcolor('blue') + "</p>");
   document.write("<p>" + str.fontcolor('green') + "</p>");

   var pos = str.indexOf("Script");
   if (pos >= 0)
   {
      document.write("Script encontrado en la posición: ");
      document.write(pos + "<br>");
   }
   else
   {
      document.write("¡Script no encontrado!" + "<br>");
   }

   document.write(str.substr(2,6));
   document.write("<br><br>");
   document.write(str.substring(2,6) + "<br>");

   document.write(str.toLowerCase());
   document.write("<br>");
   document.write(str.toUpperCase() + "<br>");
</script>
</body>
</html>
```

En este ejemplo podemos ver varios ejemplos del funcionamiento de las funciones de cadena que podemos encontrar en la clase *String.* La propiedad **length**, como en *Array*, nos devuelve, en este caso, el número de caracteres de la cadena de texto. **fontcolor** es una función que permite generar cadenas de distintos colores (nombres o valores hexadecimales). **indexOf** es una función que devuelve la posición de una cadena dentro de otra (partiendo de cero). Si es igual a -1, es que no se ha localizado. **substr** y **substring** funcionan extrayendo subcadenas de otras, pero con funcionamientos diferentes. *substr* nos devuelve una subcadena que comienza en el primer parámetro, devolviendo el número de caracteres especificado en el segundo parámetro. *substring* devuelve una subcadena que se comprende entre el primer y segundo parámetro (esto es, contando siempre con un índice base de 0). Por último, **toLowerCase** y **toUpperCase** devuelven la misma cadena pero convertida a minúsculas y mayúsculas, respectivamente.

A continuación se muestra una lista muy útil con los métodos y propiedades propios de la clase *String*, pero ¡cuidado! estos metodos dependen de que el navegador los implemente, o sea que no tienen porque funcionar en todos los navegadores y/o versiones.

Propiedades

- length
- prototype
- constructor

Métodos

- anchor()
- big()
- blink()
- bold()
- charAt()
- charCodeAt()
- concat()
- fixed()
- fontcolor()
- fontsize()
- fromCharCode()
- indexOf()
- italics()
- lastIndexOf()
- link()
- localeCompare()
- match()
- replace()
- search()
- slice()
- small()
- split()
- strike()
- sub()
- substr()
- substring()
- sup()
- toLocaleLowerCase()
- toLocaleUpperCase()
- toLowerCase()
- toString()
- toUpperCase()
- valueOf()

Gestión de formularios

Una de las utilidades más interesantes de JavaScript es la posibilidad de comprobar y gestionar formularios de forma que podamos incluso evitar que se envíe un formulario si los datos no son correctos. Primero estudiaremos como controlar la gestión y el envío de datos, y después nos sumergiremos en la comprobación propiamente dicha de los datos.

Gestión del formulario de datos

Cuando tenemos un formulario de datos que queremos enviar a un servidor para su correspondiente gestión por parte de una aplicación CGI, resulta muy interesante en ocasiones tener la posibilidad de comprobar la integridad de los datos antes de ser enviados al servidor. Además, tener la posibilidad de evitar el envío del formulario nos permite un mayor control sobre su gestión. Vamos el siguiente ejemplo:

```
<form action="prog.php" method="post" name="formu" id="formu"
      onsubmit="return comprobar()">
   Tu nombre: <input type="text" name="nombre" value=""><br>
   Tu edad:   <input type="text" name="edad" value="" size="2" maxlength="2"><br>
              <input type="submit" value="   Enviar   ">
</form>
```

Podemos apreciar en el formulario que hemos añadido un atributo nuevo, el evento **onsubmit**, que permite llamar a una función creada por nosotros para comprobar los datos que vamos a enviar. La expresión

```
return comprobar()
```

hará que el contenido de *onsubmit* sea *true* o *false*, dependiendo de lo que devuelva la funcion *comprobar()*. Si el valor es *true*, los datos se enviarán al servidor, y si es *false*, se retendrá la ejecución del formulario. A continuación estudiaremos una posible función que podemos usar en este caso:

```
<script language="javascript">
function comprobar()
{
   var nombre = document.formu.nombre.value;
   var edad = document.formu.edad.value;

   if (nombre.length > 30)
   {
      alert("Tu nombre es demasiado grande. Redúcelo.");
      return false;
   }

   if (edad >= 20 && edad <= 40)
   {
      alert("Si tienes entre 20 y 40 años, no puedes usar este programa.");
      return false;
   }

   return true;
}
</script>
```

Este script lo podemos colocar tanto en <head> como en <body>. Como estudiaremos ahora, realiza una serie de comprobaciones automáticas que impedirán o no la ejecución y envío del formulario.

Lo primero que hacemos es definir dos variables que contendrán el valor de los dos controles del formulario que queremos controlar. Para comprender esto, estudiemos las expresiones:

```
var nombre = document.formu.nombre.value;
```

Definimos la variable *nombre*, que será igual a la expresión "valor del control nombre del formulario formu del objeto de JavaScript document". Es decir, podemos acceder a cualquier parte del documento por medio de sus nombres (o identificadores, según el navegador), y concretamente a los componentes de un formulario de la forma en que aparece en el ejemplo. Así, una vez que hemos obtenido los valores, podemos procesar esos valores, como aparece en el ejemplo.

Un detalle a tener en cuenta es que en el caso de que efectivamente la situación dé lugar a un error (en este ejercicio, hemos realizado las condiciones para que sean ciertas si efectivamente se produce un error en los datos), la función devolverá *false*, lo que bloqueará el envío del formulario al servidor. Sólo si se superan las dos pruebas que ponemos a los datos, se devolverá *true* y el formulario se enviará. El código completo para este programa sería:

```
<html>
<head>
<script language="javascript">
function comprobar()
{
   var nombre = document.formu.nombre.value;
   var edad = document.formu.edad.value;

   if (nombre.length > 30)
   {
      alert("Tu nombre es demasiado grande. Redúcelo.");
      return false;
   }

   if (edad >= 20 && edad <= 40)
   {
      alert("Si tienes entre 20 y 40 años, no puedes usar este programa.");
      return false;
   }

   return true;
}
</script>
</head>
<body>
<form action="prog.php" method="post" name="formu" id="formu"
      onsubmit="return comprobar()">
   Tu nombre: <input type="text" name="nombre" value=""><br>
   Tu edad:   <input type="text" name="edad" value="" size="2" maxlength="2"><br>
              <input type="submit" value="   Enviar   ">
</form>
</body>
</html>
```

Vamos a estudiar una serie de casos particulares que se alejan de alguna forma de la forma de obtener valores que hemos visto en el ejemplo anterior, o alternativas sobre como comprobar los datos. En el siguiente ejemplo vemos como capturar el contenido de una serie de "radios", ya que podemos usar una función específica para comprobar que efectivamente hemos pulsado uno de los "radios":

```
<html>
<head>
<script language="javascript">
function check(browser)
{
   document.formu.respuesta.value = browser;
}
</script>
</head>

<body>
<form name="formu" id="formu">
   Selecciona tu navegador favorito:<br><br>
   <input type="radio" name="browser" onclick="check(this.value)"
          value="Internet Explorer"> Internet Explorer<br>
   <input type="radio" name="browser" onclick="check(this.value)"
          value="Netscape"> Netscape<br>
   <input type="radio" name="browser" onclick="check(this.value)"
          value="Opera"> Opera<br>
   <br>
   <input type="text" name="respuesta" size="20">
</form>
</body>
</html>
```

Fijémonos en las etiquetas <input>: el evento **onclick** contiene una llamada a la función *check*, cuyo parámetro es una cadena de texto. En este caso la cadena de texto es conseguida por medio de la expresion **this.value**. ¿Por qué *this.value*? Ya conocemos lo que significa *this*, es la llamada al propio objeto, en este caso el "radio", y con esta expresión leemos el valor de la etiqueta <input>. Con el objeto por defecto *this*, podemos acceder a cualquer propiedad y función de la etiqueta en la que nos encontramos en sus códigos de eventos.

En el próximo ejemplo veremos como procesar los checkboxes de un formulario. Debemos tener en cuenta que cada grupo de checkboxes pueden tener el mismo nombre, por lo que el acceso a un grupo de estas etiquetas debe ser realizado por medio de una matriz, como veremos ahora. Además, no accederemos a la propiedad value para determinar cual está pulsada, si no a la propiedad checked, que es un valor booleano que nos indicará si efectivamente un checkbox concreto está pulsado o no.

```
    <html>

    <head>
    <script type="text/javascript" >
    function check()
    {
            cafe=document.formu.cafe;
            respuesta=document.formu.respuesta;
            txt="";
            for (i=0;i<cafe.length;++i)
            {
                    if (cafe[i].checked)
                    {
                            txt=txt + cafe[i].value + " ";
                    }
            }

            respuesta.value="Tu quieres café con "  + txt;
    }
    </script>
    </head>

    <body>
    <form name="formu" id="formu">
    ¿Cómo quieres tu café? &lt;br&gt;&lt;br&gt;
    <input type="checkbox" name="cafe" value="crema">Con crema&lt;br&gt;
    <input type="checkbox" name="cafe" value="azúcar">Con azúcar&lt;br&gt;
    &lt;br&gt;
    <input type="button" name="test" onclick="check()" value="Enviar pedido" >
    &lt;br&gt;&lt;br&gt;
    <input type="text" name="respuesta" size="50">
    </form>
    </body>

    </html>
```

Como podemos ver, hay varias cosas a tener consideración: los checkboxes, al tener el mismo nombre (podrían tenerlo distinto, pero este ejemplo nos permite ver esta posibilidad), se agrupan en forma de matriz, y por lo tanto, debemos recorrer la lista de controles para acceder a sus contenidos. En nuestro caso, accedemos a la propiedad checked, que nos permite saber si el checkbox está pulsado (true) o no (false). En la variable "txt" acumulamos los valores de los checkboxes (a los que accedemos por medio del correspondiente índice, como ya sabemos), y después mostramos el resultado en la linea de texto que hay más abajo. Al introducir el texto en la propiedad value, cambiamos también el contenido del campo de texto.

En el siguiente ejemplo veremos como averiguar el dato pulsado en una lista de datos:

```
<html>
<head>
<script type="text/javascript" >
function cual()
{
        txt=document.formu.lista.options[document.formu.lista.selectedIndex].text;
        document.formu.favorito.value=txt;
}
</script>
</head>

<body>
<form id="formu" name="formu">
Elige tu navegador favorito:
<select name="lista" onchange="cual()">
        <option>Internet Explorer</option>
        <option>Netscape</option>
        <option>Opera</option>
</select>
&lt;br&gt;&lt;br&gt;
Tu navegador favorito es:  <input type="text" name="favorito" size="20">
</form>
</body>

</html>
```

La expresión

```
document.formu.lista.options[document.formu.lista.selectedIndex]
```

nos permite acceder a una de las opciones (es una matriz) de la lista de datos. En este caso, como las opciones no tienen valor, podemos acceder a la cadena de caracteres de la opción por medio de la propiedad text. En concreto, podemos conocer qué elemento se ha seleccionado por medio de la propiedad selectedIndex del control de lista. Sólo un detalle más a tener en cuenta: por un defecto de los navegadores, no es posible elegir sin más el elemento actualmente seleccionado en la lista, ya que no producirá ningún evento. Hay que tener esto en consideración para evitar posibles problemas a la hora de trabajar con las listas. Un truco para subsanar este inconveniente es poner un primer option en blanco (<option> </option>) que será el que aparezca al cargarse el select.

Apéndice A. Como escribir código en JavaScript

En este apéndice aprenderemos a insertar código JavaScript en nuestras páginas. Como sabemos, las páginas web se componen de código HTML (HyperText Markup Language), y para incluir el código Javascript utilizamos una marca HTML, <script>. Esta marca puede encontrarse en cualquier parte de la página web, tanto en el <head> como en el <body>. Aunque <script> es la forma más corta existen otras formas de definir código script. Por ejemplo <script language="JavaScript">. W3C recomienda utilizar el tag <script type="text/javascript">.

```
<html>
<head>
<title>Esta es una pagina web</title>
<script type="text/javascript">
        var mi_numero = 1;

        function calcula(numero) {
                return numero + mi_numero;
        }
</script>
</head>
<body>
<script>
        document.write(calcula(1));
</script>
</body>
</html>
```

Este ejemplo mostrará un numero '2' en el navegador

Además, podemos especificarle el lenguaje en el que queremos programar. Existen otros lenguajes para navegador como Visual Basic Script y PerlScript, pero nosotros usamos Javascript porque es universal: todos los navegadores lo soportan, mientras que los otros dependen de la plataforma donde nos encontremos. Para indicar el lenguaje, podemos escribir lo siguiente:

```
<script language="Javascript">

</script>
```

De esta forma indicamos el lenguaje a usar. Esto es necesario en el caso de que tengamos que usar lenguajes combinados en la misma página, como en el caso de que queramos enlazar una película flash con nuestra pagina web.

Otra forma de escribir Javascript en una página web es utilizando los eventos de las etiquetas HTML. Las etiquetas HTML tienen varios "eventos" que responden a determinados sucesos, como por ejemplo el click del ratón, el envío de un formulario, o la carga de una página. Por ejemplo, si queremos que aparezca un mensaje al cargar la página que estamos viendo, haríamos algo como esto:

```
<html>
<head>
</head>

<body onload="alert('Hola, esto es una página web')">
texto
</body>
</html>
```

Esto hará que aparezca un mensaje nada más cargar la página web. También podemos aplicar estos eventos como enlaces, botones, imágenes, etc... Prácticamente cualquier etiqueta HTML soporta eventos como onclick, que permite responder a una pulsación del botón izquierdo del ratón.

```
<html>
<head>
</head>

<body>
<a href="http://www.google.com/ " onclick="alert('Vas a ir a Google')">Google</a>
</body>
</html>
```

En este ejemplo vemos cómo al mismo tiempo que vamos a Google, el navegador nos avisa de lo que vamos a hacer antes de que ocurra. Este tipo de acciones se pueden usar para comprobar formularios antes de enviar los datos (e incluso, evitar su envío si no son correctos), comprobar dónde pinchamos en una imagen, etc..., observando los cambios en los objetos Javascript.

Y una última manera de ejecutar código Javascript es adjuntando un archivo al código principal, de tal forma que podemos agrupar las funciones, clases y demás en un archivo, y reutilizar ese archivo tantas veces como queramos posteriormente. Un ejemplo puede ser éste:

funciones.js:

```
function saludo(nombre) {
        alert('Hola, ' + nombre);
}
```

saludo.html:

```
<html>
<head>
<title>Esta es una pagina web</title>
<script language="Javascript" src="funciones.js"></script>
</head>
<body>
<script>
        saludo('Ana');
</script>
</body>
</html>
```

En este ejemplo vemos cómo podemos incluir un código Javascript desde otro archivo y utilizar las funciones incluidas dentro de nuestro código, en tantos archivos como queramos. De esta forma podemos reutilizar el código todo lo necesario.

Los comentarios

Los comentarios son uno de los elementos más despreciados en todos los lenguajes de programación, pero son de suma utilidad: permiten aclarar y sintetizar el código, además de servir de eventuales "ocultadores" de código, ya que todo lo que se encuentra en un comentario no es interpretado por el navegador. Podemos escribir comentarios de dos formas diferentes:

```
// Este es un comentario de una línea
```

Ponemos dos barras normales para crear un comentario de una línea. Este comentario también lo podemos usar en el caso de que queramos ocultar una línea concreta de la ejecución del programa. Si queremos realizar un comentario de múltiples líneas, haremos:

```
/* Este
es un
comentario
de
múltiples
líneas */
```

De esta forma, podemos comentar varias líneas de texto o código en un bloque. Esto es bastante interesante cuando tenemos que ocultar una gran cantidad de código continuo, que de otra forma tendríamos que comentar línea a línea con las dos barras. var input1 = document.getElementById("form1_input1");

```
var input2    = document.getElementById("form1_input2");
var sInput    = input1.value + "," + input2.value;
var divSalida = document.getElementById("div1");
    divSalida.innerHTML = sInput;
```

Los comentarios en el código son muy útiles para ayudar a comprender el sentido del programa, tanto para el creador del código como para otros que puedan leerlo.

Los nombres de las variables

Javascript es un lenguaje no tipado, es decir una variable puede servir para almacenar un número, una cadena de texto, un elemento DOM de HTML (sería más exacto hablar de "una referencia a un elemento DOM de HTML"). Este característica puede ser una ventaja cuando se tiene cierta soltura con el lenguaje, sin embargo puede ser una trampa mortal para aquellos que empiezan, ya que es fácil que intenten obtener el valor del texto almacenado en la variable "campo1" cuando en realidad "campo1" no contiene un número sino que referencia a un elemento de formulario input con identificador "campo1", de forma que para obtener el texto del campo en realidad deberían consultar el valor de "campo1.value". Estos errores son frecuentes y es fácil evitarlos si a la hora de elegir el nombre de nuestras variables prefijamos las mismas indicando el tipo de elemento al que referencian. P.ej, una variable que almacene texto puede ir prefijada con "s" (de string), una referencia a un objeto div puede ir prefijado con el propio "div" y una referencia a un input de formulario puede prefijarse con input. Así por ejemplo escribiríamos:

```
var input1    = document.getElementById("form1_input1");
var input2    = document.getElementById("form1_input2");
var sInput    = input1.value + "," + input2.value;
var divSalida = document.getElementById("div1");
    divSalida.innerHTML = sInput;
```

También es conveniente adjuntar un prefijo a las variables globales que van a ser compartidas por todas las funciones del script de forma que no se confundan con variables locales de igual nombre y sea fácil entender que se trata de variables globales compartidas sin necesidad de buscar su definición original cuando estamos inspeccionando el código de una función donde se utiliza. Por ejemplo:

```
function muestraMensaje(sMensaje){
    globDivSalida.innerHTML = this.sMensaje;
}
```

En el anterior ejemplo se entiende que globDivSalida es una variable global (que probablemente habrá sido inicializada inmediatamente al arrancar el script) gracias a su prefijo glob y que además referencia a un div, gracias a su prefijo Div

Apéndice B. Operadores en JavaScript

Los operadores son uno de los elementos fundamentales en cualquier lenguaje de programación, ya que son los que nos permiten trabajar con variables y datos. A continuación vamos a estudiar los distintos operadores divididos en categorías:

Operadores aritméticos:

Operador	Significado	Ejemplo
+	suma	números y cadenas
-	resta	
*	producto	
/	división	
%	módulo (resto)	20 % 10 (= 0)
++	suma tipográfica	variable++; ++variable; (variable = variable + 1)
-- (dos guiones)	resta tipográfica	variable--; --variable; (variable = variable - 1)

Operadores de asignación:

Operador	Significado	Ejemplo	Es igual a
=	Asignación de datos	x = 1;	
+=	Asignación y suma	x += 1;	x = x + 1;
-=	Asignación y resta	x -= 1;	x = x - 1;
*=	Asignación y producto	x *= 1;	x = x * 1;
/=	Asignación y división	x /= 1;	x = x / 1;
%=	Asignación y módulo	x %= 1;	x = x % 1;

Operadores condicionales (comparativos):

Operador	Significado	Ejemplo
==	es igual a	5 == 8 es falso
!=	no es igual a	5 != 1 es verdad
>	es mayor que	5 > 1 es verdad
<	es menor que	5 < 8 es verdad
>=	es mayor o igual que	5 >= 8 es falso
<=	es menor o igual que	5 <= 1 es falso

Operadores lógicos:

Operador	Significado	Ejemplo
&&	Y	1 == 1 && 2 < 1 es falso
||	O	1 == 2 || 15 > 2 es verdad
!	NO	!(1 > 2) es verdad

Suma y Resta Tipográfica

La diferencia en el uso de la suma y resta tipográfica radica en el momento en que se realiza la operación. Cuando los símbolos preceden a la variable (ej: ++a; --a;) la operación se realiza antes de ejecutar el resto de las operaciones en la linea. Cuando la variable precede a los símbolos (ej: a++; a--;) la operación se realiza despues de ejecutar el resto de las operaciones en la linea. En caso de que la linea de código no contenga ninguna operación extra el resultado será el mismo. Aquí algunos ejemplos para mostrar las diferencias:

```
var a=3;
var b=7;
a++;
--b;
//a=4, b=6
var c = b - a++; //Resultados: c = 1 (resta), luego a = 5 (++)
//a=5, b=6, c=1
c = --b / a++;   //Resultados: b = 5 (--), luego c = 1 (división), luego a = 6
```

Apéndice C. Ejemplos de código

En un programa JavaScript nos podemos encontrar con dos elementos básicos: **código** y **datos**. La parte del código es la que nos permite hacer cosas dentro de la página web, y la parte de datos es la que define el estado de la página web en un momento determinado. Los datos de un programa se guardan dentro de **variables**.

¿Qué es una variable? Una variable es como una caja: dentro de una caja nosotros podemos guardar cosas. Sólo que en las variables de JavaScript la caja sólo puede guardar una sola cosa a la vez. ¿Y por qué se las llama variables? Se las denomina así porque su contenido puede cambiar en cualquier momento durante el desarrollo del programa. De esta forma, una variable en JavaScript puede contener distintas cosas según donde se encuentre en el programa: números, letras, etc...

Tipos de datos en JavaScript

Existen cuatro tipos básicos:

- **Números** (enteros, decimales, etc.).
- **Cadenas de caracteres**
- **Valores lógicos** (True y False).
- **Objetos** (una ventana, un texto, un formulario, etc.).

Declaración de variables

A continuación aparece un ejemplo de declaración de una variable en JavaScript:

```
var miVar = 1234;
```

Aquí hemos definido una variable numérica con un valor entero. Pero también podríamos definir un número con decimales:

```
var miVar = 12.34;
```

Como podemos ver, la nueva variable usa el operador '.' (punto) para distinguir entre la parte entera y la parte decimal. Esto es importante, porque no podemos usar la coma como hacemos en España.

Si queremos definir una cadena de caracteres, lo podemos hacer de la siguiente manera:

```
var miCadena = 'Hola, mundo';
// o bien:
var miCadena = "Hola, mundo";
```

Aquí vemos que podemos usar los dos tipos de comillas para crear cadenas de caracteres, y será muy útil cuando trabajemos con ello (podemos incluso combinarlas dentro de la misma cadena).

También podemos crear variables con valores lógicos. Eso significa que la variable podrá tener sólo dos valores: verdad o mentira.

```
var miVar = true;
```

Los valores admitidos para este tipo de variables son *true* y *false*. Este tipo de variables nos vendrán muy bien para crear condiciones y como valor para devolver en funciones, que veremos más adelante.

Y por último tenemos un tipo de dato especial: los objetos. ¿Y qué son los objetos? Son "cosas" que podemos usar en nuestro programa para representar "entidades". Esto lo entenderemos muy fácilmente con unos ejemplos.

Estamos rodeados de objetos: mesas, libros, monitores, ratones, cuadros, etc... Algunos son más simples y otros son más complicados. Podemos manipular todos ellos según sus características y su forma de interactuar con el entorno donde están. Por ejemplo, una mesa sabemos que tiene cuatro patas, una tabla lisa, y que es de un color o varios colores. Es decir, que podemos determinar una mesa por sus propiedades o atributos. Pero además, con la mesa podemos hacer cosas: podemos poner cosas encima, podemos usarla para comer o leer y a veces podemos colgar cosas de ellas, por ejemplo en un revistero. Todo eso son métodos o comportamientos que la mesa tiene para interactuar con el resto de su entorno.

Pues bien, podemos decir que los objetos en JavaScript son muy parecidos: tienen **propiedades** (datos) y **métodos** (código). Podemos crear y usar objetos para manejar elementos del navegador web: una ventana del navegador es un objeto *window*, una página HTML es un objeto *document*, y una imagen es un objeto de tipo *Image*. Es fácil darse cuenta de que los objetos son de un determinado tipo: un objeto mesa, un objeto ventana, un objeto ratón, etc... Todos los objetos de un mismo tipo tienen características semejantes, aunque luego cada objeto tiene propiedades con valores distintos dependiendo de cada caso. Así, dos mesas puede tener color marrón o azul, pero las dos seguirán teniendo patas, que pueden ser 4 ó 5, depende... En JavaScript, los objetos son muy importantes, como vamos a comprobar en el siguiente capítulo, que trata de los arrays (matrices) y las sentencias de control.

Operar con variables

Como ya estudiamos en el capítulo anterior, en JavaScript podemos definir unos elementos llamados variables que nos permiten almacenar datos de distintos tipos. Naturalmente, nosotros podemos usar esos datos en nuestros programas y, como ya se indicó entonces, podemos incluso variar esos datos manteniendo la variable donde los depositamos. A esta operación se le llama modificar la variable, y es una de las bases de la programación moderna.

Las variables podemos usarlas en multitud de situaciones, al mostrar datos, al enviarlos y recibirlos, en expresiones y llamadas a funciones... Podemc tratar con variables para almacenar los datos que vamos a usar a lo largo del programa, tanto globalmente en toda la aplicacion como de forma exclusiva con las funciones que creemos, como veremos en el capítulo correspondiente.

```
var numero = 1;
numero = numero + 2;
numero += 3;
```

El resultado final de esta operación sera que la variable numero será igual a 6. En la primera línea lo que hemos hecho es declarar la variable numero con el valor inicial 1. Despues, hemos incrementado el valor de la variable con la misma variable, sumándole 2, y posteriormente hemos vuelto a incrementar la variable sumándole 3 por medio del operador tipográfico +=. Los operadores se encuentran en el primer apéndice del curso.

Sin embargo, surge un pequeño problema cuando tenemos que tratar con cantidades mayores de datos. Las variables como tales sólo nos permiten gestionar un dato cada una de ellas, con lo que cuando tenemos que gestionar grupos mayores de datos, se hace realmente complicado. Miremos el siguiente ejemplo, en el que definimos unos nombres:

```
var nombre1 = 'pepe';
var nombre2 = 'toño';
var nombre3 = 'mari';
var nombre4 = 'lucas';
var nombre5 = 'sonia';
var nombre6 = 'ruth';
var nombre7 = 'tete';
```

Si ahora quisiéramos listar estos datos (más adelante veremos cómo), tendríamos que referirnos a cada variable en concreto, con lo que tenemos pululando por nuestro programa siete variables a las que será difícil referirnos de una forma genérica (por ejemplo, como estudiaremos más adelante, para listarlos dinámicamente en un formulario). Para resolver este problema tenemos una solución: los arrays (matrices).

Arrays (Matrices)

Las matrices son variables que contienen un objeto de tipo *Array*. Podemos definir una matriz de la siguiente manera:

```
var matriz = new Array();
```

De esta forma, hemos creado una matriz vacía que puede contener un numero ilimitado de elementos, tantos como nos permita el sistema donde se ejecuta. Las matrices vienen a ser como cajas que en vez de contener una sola cosa, contienen muchas, como si pudiéramos dividir la caja en compartimentos en los cuales pudiéramos ir depositando cosas.

Además, podemos crear matrices con una "dimensión", es decir, que podemos hacer que la matriz se inicie con un número de elementos determinado:

```
var matriz = new Array(15);
```

Con esta instrucción, lo que hemos hecho es crear una matriz de quince elementos. Pero ahora lo interesante es saber cómo llamar a esos elementos, ya que si creamos la matriz, pero no sabemos operar con ella, no sirve para mucho, ¿no? La forma de acceder a un elemento de la matriz es la siguiente:

```
elemento = matriz[1];
```

En este ejemplo, la variable *elemento* contendrá el valor del elemento 1 de la matriz. Es lo que se llama **índice** de la matriz, e identifica a cualquiera de los elementos de la matriz. Hay que fijarse en que utilizamos los corchetes "[]" para señalar un elemento de la matriz. El primer elemento de la matriz es el de índice '0', no el de índice '1'. Así, para el anterior ejemplo de una matriz de 15 elementos, el último elemento posible es el 14.

De la misma forma, podemos dar un valor a cualquiera de los elementos de la matriz:

```
matriz[5] = 'hola';
```

Hemos asignado el valor *hola* al elemento 5 de la matriz. Los elementos de una matriz pueden contener cualquier tipo de dato, y se pueden cambiar en cualquier parte del programa, al igual que ocurre con las variables.

¿Y si queremos saber cuántos datos tenemos en la matriz? Como dijimos antes, las matrices son objetos de tipo Array, y los objetos pueden tener atributos (datos) y funciones (código). El atributo que debemos usar con matrices es **length**:

```
longitud = matriz.length;
```

De esta forma, podemos saber cuantos elementos tiene la matriz. Recordad que como el primer índice es '0', el último elemento será siempre *matriz.length - 1*.

Si necesitamos que la matriz contenga más elementos, podemos redimensionar la matriz aplicándole un nuevo objeto de matriz:

```
matriz = new Array( longitud que queramos );
```

Sin embargo, perderemos todos los elementos que tuviéramos anteriormente.
