Lógica y Estructura de Datos

JAVASCRIPT

Técnico Superior en Desarrollo de Software

Esc. Superior Nº 49 "Cap. Gral. J.J. Urquiza"

Año 2018

Ing. Álvaro Hergenreder

CAPÍTULO 1: INTRODUCCIÓN

1.1 ¿QUÉ ES JAVASCRIPT?

JavaScript es un lenguaje de programación interpretado, dialecto del estándar ECMAScript. Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.

Se utiliza principalmente en el lado del cliente, implementado como parte de un navegador web permitiendo crear interacción con el usuario y páginas web dinámicas, aunque actualmente es posible ejecutar JavaScript en el propio servidor (NodeJS). Su uso en aplicaciones externas a la web, por ejemplo en documentos PDF o aplicaciones de escritorio (mayoritariamente widgets) es también significativo.

JavaScript se diseñó con una sintaxis similar al lenguaje de programación C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo Java y JavaScript no están relacionados y tienen semánticas y propósitos diferentes.

1.2 BREVE HISTORIA

A principios de los años 90, la mayoría de usuarios que se conectaban a Internet lo hacían con módems a una velocidad máxima de 28.8 kbps. En esa época, empezaban a desarrollarse las primeras aplicaciones web y por tanto, las páginas web comenzaban a incluir formularios complejos.

Con unas aplicaciones web cada vez más complejas y una velocidad de navegación tan lenta, surgió la necesidad de un lenguaje de programación que se ejecutara en el navegador del usuario. De esta forma, si el usuario no rellenaba correctamente un formulario, no se le hacía esperar mucho tiempo hasta que el servidor volviera a mostrar el formulario indicando los errores existentes.

Brendan Eich, un programador que trabajaba en Netscape, pensó que podría solucionar este problema adaptando otras tecnologías existentes (como ScriptEase) al navegador Netscape Navigator 2.0, que iba a lanzarse en 1995. Inicialmente, Eich denominó a su lenguaje LiveScript.

Posteriormente, Netscape firmó una alianza con Sun Microsystems para el desarrollo del nuevo lenguaje de programación. Además, justo antes del lanzamiento Netscape decidió cambiar el nombre por el de JavaScript. La razón del cambio de nombre fue exclusivamente por marketing, ya que Java era la palabra de moda en el mundo informático y de Internet de la época.

La primera versión de JavaScript fue un completo éxito y Netscape Navigator 3.0 ya incorporaba la siguiente versión del lenguaje, la versión 1.1. Al mismo tiempo, Microsoft lanzó JScript con su navegador Internet Explorer 3. JScript era una copia de JavaScript al que le cambiaron el nombre para evitar problemas legales.

Para evitar una guerra de tecnologías, Netscape decidió que lo mejor sería estandarizar el lenguaje JavaScript. De esta forma, en 1997 se envió la especificación JavaScript 1.1 al organismo ECMA (European Computer Manufacturers Association).

ECMA creó el comité TC39 con el objetivo de "estandarizar de un lenguaje de script multiplataforma e independiente de cualquier empresa". El primer estándar que creó el comité TC39 se denominó **ECMA-262**, en el que se definió por primera vez el lenguaje ECMAScript.

Por este motivo, algunos programadores prefieren la denominación ECMAScript para referirse al lenguaje JavaScript. De hecho, JavaScript no es más que la implementación que realizó la empresa Netscape del estándar ECMAScript.

La organización internacional para la estandarización (ISO) adoptó el estándar ECMA-262 a través de su comisión IEC, dando lugar al estándar ISO/IEC-16262.

1.3 ESPECIFICACIONES OFICIALES

ECMA ha publicado varios estándares relacionados con ECMAScript. En Junio de 1997 se publicó la primera edición del estándar ECMA-262. Un año después, en Junio de 1998 se realizaron pequeñas modificaciones para adaptarlo al estandar ISO/IEC-16262 y se creó la segunda edición.

La sexta edición del estándar ECMA-262 (publicada en junio de 2015) es la versión que utilizan los navegadores actuales y se puede consultar gratuitamente en http://www.ecma-international.org/publications/standards/Ecma-262.htm

1.4 CÓMO INCLUIR JAVASCRIPT EN DOCUMENTOS HTML

La integración de JavaScript y HTML es muy flexible, ya que existen al menos tres formas para incluir código JavaScript en las páginas web.

1.4.1 INCLUIR JAVASCRIPT EN EL MISMO DOCUMENTO HTML

El código JavaScript se encierra entre etiquetas <script> y se incluye en cualquier parte del documento. Aunque es correcto incluir cualquier bloque de código en cualquier zona de la página, se recomienda definir el código JavaScript dentro de la cabecera del documento (dentro de la etiqueta <head>):

```
<!DOCTYPE html>
<head>
     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
     <title>Ejemplo de código JavaScript en el propio documento</title>
     <script type="text/javascript">
       document.write ("Un mensaje de prueba");
     </script>
</head>
<body>
     <p>Un párrafo de texto</p>
</body>
</html>
```

Para que la página HTML resultante sea válida, hasta hace poco tiempo era necesario añadir el atributo type a la etiqueta <script>. Los valores que se incluyen en el atributo type están estandarizados y para el caso de JavaScript, el valor correcto es text/javascript. Actualmente no es necesario hacerlo, solo con <script> …. </script> es suficiente.

Este método se emplea cuando se define un bloque pequeño de código o cuando se quieren incluir instrucciones específicas en un determinado documento HTML que completen las instrucciones y funciones que se incluyen por defecto en todos los documentos del sitio web.

El principal inconveniente es que si se quiere hacer una modificación en el bloque de código, es necesario modificar todas las páginas que incluyen ese mismo bloque de código JavaScript.

1.4.2 DEFINIR JAVASCRIPT EN UN ARCHIVO EXTERNO

Las instrucciones JavaScript se pueden incluir en un archivo externo de tipo JavaScript que los documentos HTML enlazan mediante la etiqueta <script>. Se pueden crear todos los archivos JavaScript que sean necesarios y cada documento HTML puede enlazar tantos archivos JavaScript como necesite.

Ejemplo:

Archivo `codigo.js`

```
document.write("Un mensaje de prueba");
```

Documento HTML

```
<!DOCTYPE html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Ejemplo de código JavaScript en el propio documento</title>
    <script type="text/javascript" src="/js/codigo.js"></script>
</head>
<body>
    <p>Un párrafo de texto</p>
</body>
</html>
```

Además del atributo type, este método requiere definir el atributo src, que es el que indica la URL correspondiente al archivo JavaScript que se quiere enlazar. Cada etiqueta <script>solamente puede enlazar un único archivo, pero en una misma página se pueden incluir tantas etiquetas <script> como sean necesarias.

Los archivos de tipo JavaScript son documentos normales de texto con la extensión .js, que se pueden crear con cualquier editor de texto como Notepad, Wordpad, EmEditor, UltraEdit, Vi, etc.

La principal ventaja de enlazar un archivo JavaScript externo es que se simplifica el código HTML de la página, que se puede reutilizar el mismo código JavaScript en todas las páginas del sitio web y que cualquier modificación realizada en el archivo JavaScript se ve reflejada inmediatamente en todas las páginas HTML que lo enlazan.

1.4.3 INCLUIR JAVASCRIPT EN LOS ELEMENTOS HTML

Este último método es el menos utilizado, ya que consiste en incluir instrucciones JavaScript dentro del código HTML de la página:

```
<!DOCTYPE>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
/>
    <title>Ejemplo de código JavaScript en el propio documento</title>
```

```
</head>
<body>
    <p onclick="alert('Un mensaje de prueba')">Un párrafo de texto</p>
</body>
</html>
```

El mayor inconveniente de este método es que ensucia innecesariamente el código HTML de la página y complica el mantenimiento del código JavaScript. En general, este método sólo se utiliza para definir algunos eventos y en algunos otros casos especiales, como se verá más adelante.

1.5 ETIQUETA NOSCRIPT

Algunos navegadores no disponen de soporte completo de JavaScript, otros navegadores permiten bloquearlo parcialmente e incluso algunos usuarios bloquean completamente el uso de JavaScript porque creen que así navegan de forma más segura.

En estos casos, es habitual que si la página web requiere JavaScript para su correcto funcionamiento, se incluya un mensaje de aviso al usuario indicándole que debería activar JavaScript para disfrutar completamente de la página. El siguiente ejemplo muestra una página web basada en JavaScript cuando se accede con JavaScript activado y cuando se accede con JavaScript completamente desactivado.

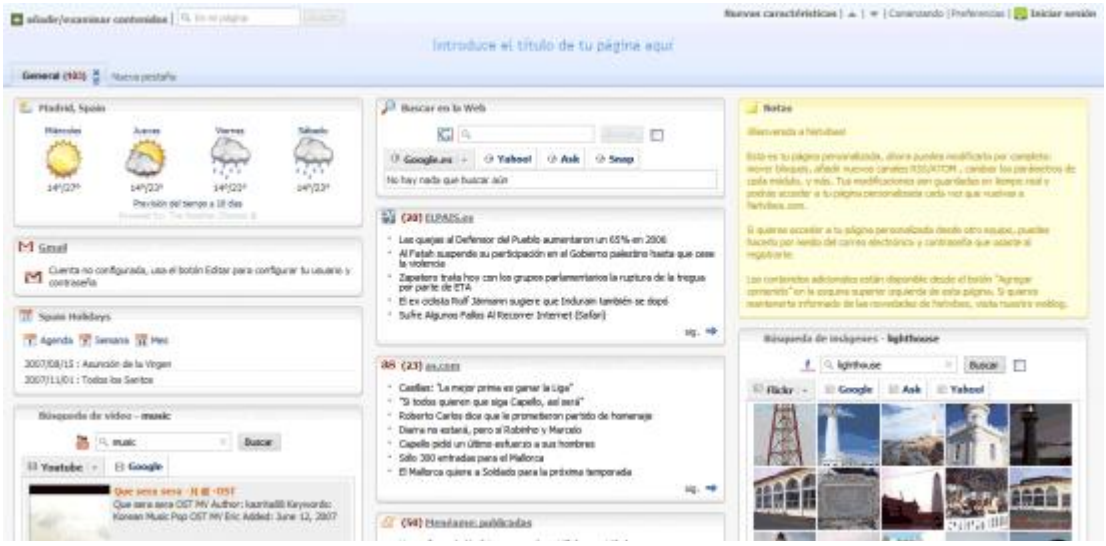

Figura 1.1 Imagen de www.netvibes.com con JavaScript activado

Figura 1.2 Imagen de www.netvibes.com con JavaScript desactivado

El lenguaje HTML define la etiqueta <noscript> para mostrar un mensaje al usuario cuando su navegador no puede ejecutar JavaScript. El siguiente código muestra un ejemplo del uso de la etiqueta <noscript>:

```
<head> ... </head>
<body>
    <noscript>
      <p>Bienvenido a Mi Sitio</p>
      <p>La página que estás viendo requiere para su funcionamiento
        El uso de JavaScript. Si lo has deshabilitado
        intencionadamente, por favor vuelve a activarlo.</p>
    </noscript>
</body>
```

La etiqueta <noscript> se debe incluir en el interior de la etiqueta <body> (normalmente se incluye al principio de <body>). El mensaje que muestra <noscript> puede incluir cualquier elemento o etiqueta HTML.

1.6 GLOSARIO BÁSICO

- **Script**: cada uno de los programas, aplicaciones o trozos de código creados con el lenguaje de programación JavaScript. Unas pocas líneas de código forman un script y un archivo de miles de líneas de JavaScript también se considera un script.

- **Sentencia**: cada una de las instrucciones que forman un script.

- **Palabras reservadas**: son las palabras (en inglés) que se utilizan para construir las sentencias de JavaScript y que por tanto no pueden ser utilizadas libremente. A continuación se indica el

listado de palabras reservadas en JavaScript, y que no podremos utilizar para nombrar nuestras variables, parámetros, funciones, operadores o etiquetas:

```
abstract
boolean break byte
case catch char class const continue
debugger default delete do double
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while with
```

1.7 POSIBILIDADES Y LIMITACIONES

Desde su aparición, JavaScript siempre fue utilizado de forma masiva por la mayoría de sitios de Internet. La aparición de Flash disminuyó su popularidad, ya que Flash permitía realizar algunas acciones imposibles de llevar a cabo mediante JavaScript. Sin embargo, la aparición de las aplicaciones AJAX y JQUERY programadas con JavaScript le ha devuelto una popularidad sin igual dentro de los lenguajes de programación web,.

JavaScript fue diseñado de forma que se ejecutara en un entorno muy limitado que permitiera a los usuarios confiar en la ejecución de los scripts. De esta forma, los scripts de JavaScript no pueden comunicarse con recursos que no pertenezcan al mismo dominio desde el que se descargó el script. Los scripts tampoco pueden cerrar ventanas que no hayan abierto esos mismos scripts. Las ventanas que se crean no pueden ser demasiado pequeñas ni demasiado grandes ni colocarse fuera de la vista del usuario (aunque los detalles concretos dependen de cada navegador).

Además, los scripts no pueden acceder a los archivos del ordenador del usuario (ni en modo lectura ni en modo escritura) y tampoco pueden leer o modificar las preferencias del navegador.

Por último, si la ejecución de un script dura demasiado tiempo (por ejemplo por un error de programación) el navegador informa al usuario de que un script está consumiendo demasiados recursos y le da la posibilidad de detener su ejecución.

A pesar de todo, existen alternativas para poder saltarse algunas de las limitaciones anteriores. La alternativa más utilizada y conocida consiste en firmar digitalmente el script y solicitar al usuario el permiso para realizar esas acciones.

CAPÍTULO 2: EL PRIMER SCRIPT

A continuación, se muestra un primer script sencillo pero completo:

```
<!DOCTYPE html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>El primer script</title>
    <script type="text/javascript">
      alert("Hola Mundo!");
    </script>

</head>
<body>
    <p>Esta página contiene el primer script</p>
</body>
</html>
```

En este ejemplo, el script se incluye como un bloque de código dentro de una página HTML. Por tanto, en primer lugar se debe crear una página HTML correcta que incluya la declaración del DOCTYPE, las secciones <head> y <body>, la etiqueta <title>, etc.

Aunque el código del script se puede incluir en cualquier parte de la página, se recomienda incluirlo en la cabecera del documento, es decir, dentro de la etiqueta <head>.

A continuación, el código JavaScript se debe incluir entre las etiquetas <script>...</script>. Además, en este caso se definió el atributo type de la etiqueta<script>. Técnicamente, el atributo type se corresponde con "el tipo MIME", que es un estándar para identificar los diferentes tipos de contenidos. El "tipo MIME" correcto para JavaScript es text/javascript.

Una vez definida la zona en la que se incluirá el script, se escriben todas las sentencias que forman la aplicación. Este primer ejemplo es tan sencillo que solamente incluye una sentencia: alert("Hola Mundo!");.

La instrucción alert() es una de las utilidades que incluye JavaScript y permite mostrar un mensaje en la pantalla del usuario. Si se visualiza la página web de este primer script en cualquier navegador, automáticamente se mostrará una ventana con el mensaje "Hola Mundo!".

A continuación se muestra el resultado de ejecutar el script:

Figura 2.1 *Mensaje mostrado con* `alert()`

CAPÍTULO 3: SINTAXIS BÁSICA

Antes de comenzar a desarrollar programas y utilidades con JavaScript, es necesario conocer los elementos básicos con los que se construyen las aplicaciones. Este capítulo explica en detalle y comenzando desde cero los conocimientos básicos necesarios para poder comprender la sintaxis básica de Javascript. En el próximo capítulo veremos aspectos más avanzados como objetos, herencia, arrays o expresiones regulares.

La sintaxis de un lenguaje de programación se define como el conjunto de reglas que deben seguirse al escribir el código fuente de los programas para considerarse como correctos para ese lenguaje de programación.

La sintaxis de JavaScript es muy similar a la de otros lenguajes de programación como Java y C. Las normas básicas que definen la sintaxis de JavaScript son las siguientes:

3.1 ESPACIOS EN BLANCO

No se tienen en cuenta los espacios en blanco y las nuevas líneas: como sucede con HTML, el intérprete de JavaScript ignora cualquier espacio en blanco sobrante, por lo que el código se puede ordenar de forma adecuada para entenderlo mejor (tabulando las líneas, añadiendo espacios, creando nuevas líneas, etc.) Sin embargo, en ocasiones estos espacios en blanco son totalmente necesarios, por ejemplo, para separar nombres de variables o palabras reservadas. Por ejemplo:

```
var that = this;
```

Aquí el espacio en blanco entre `var` y `that` no puede ser eliminado, pero el resto sí.

3.2 COMENTARIOS

JavaScript ofrece dos tipos de comentarios:

- de línea comenzando con // y
- de bloque gracias a los caracteres /* */

Ejemplo de comentario de una sola línea:

```
// a continuación se muestra un mensaje
document.write("mensaje de prueba");
```

Ejemplo de comentario de varias líneas:

```
/* Los comentarios de varias líneas son muy útiles cuando se necesita
incluir bastante información en los comentarios */
document.write ("mensaje de prueba");
```

El formato /* */ de comentarios puede causar problemas en ciertas condiciones, como en las expresiones regulares, por lo que hay que tener cuidado al utilizarlo. Por ejemplo:

```
/*
    var rm_a = /a*/.match(s);
*/
```

provoca un error de sintaxis. Por lo tanto, suele ser recomendable utilizar únicamente los comentarios de línea, para evitar este tipo de problemas.

Otras reglas …

- **Se distinguen las mayúsculas y minúsculas**: al igual que sucede con la sintaxis de las etiquetas y elementos HTML. Sin embargo, si en una página HTML se utilizan indistintamente mayúsculas y minúsculas, la página se visualiza correctamente, siendo el único problema la no validación de la página. En cambio, si en JavaScript se intercambian mayúsculas y minúsculas el script no funciona.

- **No se define el tipo de las variables:** al crear una variable, no es necesario indicar el tipo de dato que almacenará. De esta forma, una misma variable puede almacenar diferentes tipos de datos durante la ejecución del script.

- **No es necesario terminar cada sentencia con el carácter de punto y coma (;):** en la mayoría de lenguajes de programación, es obligatorio terminar cada sentencia con el carácter ;. Aunque JavaScript no obliga a hacerlo, es conveniente seguir la tradición de terminar cada sentencia con el carácter del punto y coma (;).

3.3 VARIABLES

Las variables en JavaScript se crean mediante la palabra reservada var. De esta forma, podemos declarar variables de la siguiente manera:

```
var numero_1 = 3;
var numero_2 = 1;
var resultado = numero_1 + numero_2;
```

La palabra reservada var solamente se debe indicar al declarar por primera vez la variable. Cuando se utilizan las variables en el resto de instrucciones del script, solamente es necesario indicar su nombre. En otras palabras, en el ejemplo anterior sería un error indicar lo siguiente:

```
var numero_1 = 3;
var numero_2 = 1;
var resultado = var numero_1 + var numero_2;
```

En JavaScript no es obligatorio inicializar las variables, ya que se pueden declarar por una parte y asignarles un valor posteriormente. Por tanto, el ejemplo anterior se puede rehacer de la siguiente manera:

```
var numero_1;
var numero_2;

numero_1 = 3;
numero_2 = 1;

var resultado = numero_1 + numero_2;
```

Una de las características más sorprendentes de JavaSript para los programadores habituados a otros lenguajes de programación es que tampoco es necesario declarar las variables. En otras palabras, se pueden utilizar variables que no se han definido anteriormente mediante la palabra reservada var. El ejemplo anterior también es correcto en JavaScript de la siguiente forma:

```
var numero_1 = 3;
var numero_2 = 1;
resultado = numero_1 + numero_2;
```

La variable **resultado** no está declarada, por lo que JavaScript crea una variable global (más adelante se verán las diferencias entre variables locales y globales) y le asigna el valor correspondiente. De la misma forma, también sería correcto el siguiente código:

```
numero_1 = 3;
numero_2 = 1;
resultado = numero_1 + numero_2;
```

En cualquier caso, se recomienda declarar todas las variables que se vayan a utilizar.

3.3.1 NOMBRES DE VARIABLES

El nombre de una variable también se conoce como **identificador** y debe cumplir las siguientes normas:

- Sólo puede estar formado por letras, números y los símbolos $ (dólar) y _ (guión bajo).
- El primer carácter no puede ser un número.

Por tanto, las siguientes variables tienen nombres correctos:

```
var $numero1;
var _$letra;
var $$$otroNumero;
var $_a__$4;
```

Sin embargo, las siguientes variables tienen identificadores incorrectos:

```
var 1numero;        // Empieza por un número
var numero;1_123;   // Contiene un carácter ";"
```

3.3.2 TIPOS DE VARIABLES

JavaScript divide los distintos tipos de variables en dos grupos: tipos primitivos y tipos de referencia o clases.

3.3.2.1 TIPOS PRIMITIVOS

JavaScript define cinco tipos primitivos: number, string, boolean, undefined y null. Además de estos tipos, JavaScript define el operador typeof para averiguar el tipo de una variable.

3.3.2.1.1 NÚMEROS

En JavaScript únicamente existe un tipo de número. Internamente, es representado como un dato de 64 bits en coma flotante, al igual que el tipo de dato double en Java. A diferencia de otros

lenguajes de programación, no existe una diferencia entre un número entero y otro decimal, por lo que 1 y 1.0 son el mismo valor. Esto es significativo ya que evitamos los problemas de desbordamiento en tipos de dato *pequeños*, al no existir la necesidad de conocer el tipo de dato.

TIPOS DE NÚMEROS

Si el número es entero, se indica su valor directamente.

```
var variable1 = 10;
```

Si el número es decimal, se debe utilizar el punto (.) para separar la parte entera de la decimal.

```
var variable2 = 3.14159265;
```

Además del sistema numérico decimal, también se pueden indicar valores en el sistema octal (si se incluye un cero delante del número) y en sistema hexadecimal (si se incluye un cero y una x delante del número).

```
var variable1 = 10;
var variable_octal = 034;
var variable_hexadecimal = 0xA3;
```

JavaScript define tres valores especiales muy útiles cuando se trabaja con números. En primer lugar se definen los valores Infinity y –Infinity para representar números demasiado grandes (positivos y negativos) y con los que JavaScript no puede trabajar.

```
var variable1 = 3, variable2 = 0;
document.write (variable1/variable2);  // muestra "Infinity"
```

El otro valor especial definido por JavaScript es NaN, que es el acrónimo de *"Not a Number"*. De esta forma, si se realizan operaciones matemáticas con variables no numéricas, el resultado será de tipo NaN.

Para manejar los valores NaN, se utiliza la función relacionada isNaN(), que devuelve true si el parámetro que se le pasa no es un número:

```
var variable1 = 3;
var variable2 = "hola";
isNaN(variable1);  // false
isNaN(variable2);  // true
isNaN(variable1 + variable2);  // true
```

Por último, JavaScript define algunas constantes matemáticas que representan valores numéricos significativos:

Constante	Valor	Significado
`Math.E`	`2.718281828459045`	Constante de Euler, base de los logaritmos naturales y también llamado *número e*
`Math.LN2`	`0.6931471805599453`	Logaritmo natural de `2`
`Math.LN10`	`2.302585092994046`	Logaritmo natural de `10`
`Math.LOG2E`	`1.4426950408889634`	Logaritmo en base `2` de `Math.E`
`Math.LOG10E`	`0.4342944819032518`	Logaritmo en base `10` de `Math.E`
`Math.PI`	`3.141592653589793`	Pi, relación entre el radio de una circunferencia y su diámetro
`Math.SQRT1_2`	`0.7071067811865476`	Raíz cuadrada de ½
`Math.SQRT2`	`1.4142135623730951`	Raíz cuadrada de `2`

De esta forma, para calcular el área de un círculo de radio r, se debe utilizar la constante que representa al número Pi:

```
var area = Math.PI * r * r;
```

3.3.2.1.2 CADENAS DE TEXTO

Las variables de tipo cadena de texto permiten almacenar cualquier sucesión de caracteres, por lo que se utilizan ampliamente en la mayoría de aplicaciones JavaScript. Cada carácter de la cadena se encuentra en una posición a la que se puede acceder individualmente, siendo el primer carácter el de la posición 0.
El valor de las cadenas de texto se indica encerrado entre comillas simples o dobles:

```
var variable1 = "hola";
var variable2 = 'mundo';
var variable3 = "hola mundo, esta es una frase más larga";
```

Las cadenas de texto pueden almacenar cualquier carácter, aunque algunos no se pueden incluir directamente en la declaración de la variable. Si por ejemplo se incluye un ENTER para mostrar el resto de caracteres en la línea siguiente, se produce un error en la aplicación:

```
var variable = "hola mundo, esta es
                                una frase más larga";
```

La variable anterior no está correctamente definida y se producirá un error en la aplicación. Por tanto, resulta evidente que algunos caracteres *especiales* no se pueden incluir directamente. De la misma forma, como las comillas (doble y simple) se utilizan para encerrar los contenidos, también se pueden producir errores:

```
var variable1 = "hola 'mundo'";
var variable2 = 'hola "mundo"';
var variable3 = "hola 'mundo', esta es una "frase" más larga";
```

Si el contenido de texto tiene en su interior alguna comilla simple, se encierran los contenidos con comillas dobles (como en el caso de la variable1 anterior). Si el contenido de texto tiene en su interior alguna comilla doble, se encierran sus contenidos con comillas simples (como en el caso de la variable2 anterior). Sin embargo, en el caso de la variable3 su contenido tiene tanto comillas simples como comillas dobles, por lo que su declaración provocará un error.

Para resolver estos problemas, JavaScript define un mecanismo para incluir de forma sencilla caracteres especiales (ENTER, Tabulador) y problemáticos (comillas). Esta estrategia se denomina "mecanismo de escape", ya que se sustituyen los caracteres problemáticos por otros caracteres seguros que siempre empiezan con la barra \:

Si se quiere incluir...	Se debe sustituir por...
Una nueva línea	`\n`
Un tabulador	`\t`
Una comilla simple	`\'`
Una comilla doble	`\"`
Una barra inclinada	`\\`

Utilizando el mecanismo de escape, se pueden corregir los ejemplos anteriores:

```
var variable = "hola mundo, esta es \n una frase más larga";
var variable3 = "hola 'mundo', esta es una \"frase\" más larga";
```

3.3.2.1.3 VARIABLES DE TIPO BOOLEAN

Se trata de una variable que sólo puede almacenar uno de los dos valores especiales definidos y que representan el valor *"verdadero"* y el valor *"falso"*.

```
var variable1 = true;
```

```
var variable2 = false;
```

Los valores true y false son valores especiales, de forma que no son palabras ni números ni ningún otro tipo de valor. Este tipo de variables son esenciales para crear cualquier aplicación, tal y como se verá más adelante.
Cuando es necesario convertir una variable numérica a una variable de tipo boolean, JavaScript aplica la siguiente conversión: el **número 0** se convierte en **false** y cualquier otro número **distinto de 0** se convierte en **true**.
Por este motivo, en ocasiones se asocia el número 0 con el valor false y el número 1 con el valor true. Sin embargo, es necesario insistir en que true y false son valores especiales que no se corresponden ni con números ni con ningún otro tipo de dato.

3.3.2.1.4 VARIABLES DE TIPO UNDEFINED

El tipo undefined corresponde a las variables que han sido definidas y todavía no se les ha asignado un valor:

```
var variable1;
typeof variable1;  // devuelve "undefined"
```

3.3.2.1.5 VARIABLES DE TIPO NULL

Se trata de un tipo similar a undefined, y de hecho en JavaScript se consideran iguales (undefined == null). El tipo null se suele utilizar para representar objetos que en ese momento no existen.

```
var nombreUsuario = null;
```

3.3.2.2 CONVERSIÓN ENTRE TIPOS DE VARIABLES

JavaScript es un lenguaje de programación *"no tipado"*, lo que significa que una misma variable puede guardar diferentes tipos de datos a lo largo de la ejecución de la aplicación. De esta forma, una variable se podría inicializar con un valor numérico, después podría almacenar una cadena de texto y podría acabar la ejecución del programa en forma de variable booleana.
No obstante, en ocasiones es necesario que una variable almacene un dato de un determinado tipo. Para asegurar que así sea, se puede convertir una variable de un tipo a otro, lo que se denomina *typecasting*.
Así, JavaScript incluye un método llamado toString() que permite convertir variables de cualquier tipo a variables de cadena de texto, tal y como se muestra en el siguiente ejemplo:

```
var variable1 = true;
variable1.toString();  // devuelve "true" como cadena de texto
var variable2 = 5;
variable2.toString();  // devuelve "5" como cadena de texto
```

JavaScript también incluye métodos para convertir los valores de las variables en valores numéricos. Los métodos definidos son **parseInt()** y **parseFloat()**, que convierten la variable que se le indica en un número entero o un número decimal respectivamente.
La conversión numérica de una cadena se realiza carácter a carácter empezando por el de la primera posición. Si ese carácter no es un número, la función devuelve el valor **NaN**. Si el primer carácter es un número, se continúa con los siguientes caracteres mientras estos sean números.

```
var variable1 = "hola";
parseInt(variable1);  // devuelve NaN
var variable2 = "34";
parseInt(variable2);  // devuelve 34
var variable3 = "34hola23";
parseInt(variable3);  // devuelve 34
var variable4 = "34.23";
parseInt(variable4);  // devuelve 34
```

En el caso de **parseFloat()**, el comportamiento es el mismo salvo que también se considera válido el carácter **.** que indica la parte decimal del número:

```
var variable1 = "hola";
parseFloat(variable1);  // devuelve NaN
var variable2 = "34";
parseFloat(variable2);  // devuelve 34.0
var variable3 = "34hola23";
parseFloat(variable3);  // devuelve 34.0
var variable4 = "34.23";
parseFloat(variable4);  // devuelve 34.23
```

3.3.2.3 TIPOS DE REFERENCIA

Aunque JavaScript no define el concepto de clase, los tipos de referencia se asemejan a las clases de otros lenguajes de programación. Los objetos en JavaScript se crean mediante la palabra reservada new y el nombre de la clase que se va a instanciar. De esta forma, para crear un objeto de tipo String se indica lo siguiente (los paréntesis solamente son obligatorios cuando se utilizan argumentos, aunque se recomienda incluirlos incluso cuando no se utilicen):

```
var variable1 = new String("hola mundo");
```

JavaScript define una clase para cada uno de los tipos de datos primitivos. De esta forma, existen objetos de tipo Boolean para las variables booleanas, Number para las variables numéricas y String para las variables de cadenas de texto. Las clases Boolean, Number y String almacenan los

mismos valores de los tipos de datos primitivos y añaden propiedades y métodos para manipular sus valores.

```
var longitud = "hola mundo".length;
```

La propiedad length sólo está disponible en la clase String, por lo que en principio no debería poder utilizarse en un dato primitivo de tipo cadena de texto. Sin embargo, JavaScript convierte el tipo de dato primitivo al tipo de referencia String, obtiene el valor de la propiedad length y devuelve el resultado. Este proceso se realiza de forma automática y transparente para el programador.

En realidad, con una variable de tipo String no se pueden hacer muchas más cosas que con su correspondiente tipo de dato primitivo. Por este motivo, no existen muchas diferencias prácticas entre utilizar el tipo de referencia o el tipo primitivo, salvo en el caso del resultado del operador typeof y en el caso de la función eval(), como se verá más adelante.

La principal diferencia entre los tipos de datos es que los datos primitivos se manipulan por valor y los tipos de referencia se manipulan, como su propio nombre indica, por referencia. Los conceptos *"por valor"* y *"por referencia"* son iguales que en el resto de lenguajes de programación, aunque existen diferencias importantes (no existe por ejemplo el concepto de puntero).

Cuando un dato se manipula por valor, lo único que importa es el valor en sí. Cuando se asigna una variable por valor a otra variable, se copia directamente el valor de la primera variable en la segunda. Cualquier modificación que se realice en la segunda variable es independiente de la primera variable.

De la misma forma, cuando se pasa una variable por valor a una función (como se explicará más adelante) sólo se pasa una copia del valor. Así, cualquier modificación que realice la función sobre el valor pasado no se refleja en el valor de la variable original.

En el siguiente ejemplo, una variable se asigna por valor a otra variable:

```
var variable1 = 3;
var variable2 = variable1;

variable2 = variable2 + 5;
// Ahora variable2 = 8 y variable1 sigue valiendo 3
```

La variable1 se asigna por valor en la variable2. Aunque las dos variables almacenan en ese momento el mismo valor, son independientes y cualquier cambio en una de ellas no afecta a la otra. El motivo es que los tipos de datos primitivos siempre se asignan (y se pasan) por valor.

Sin embargo, en el siguiente ejemplo, se utilizan tipos de datos de referencia:

```
// variable1 = 25 diciembre de 2009
var variable1 = new Date(2009, 11, 25);
// variable2 = 25 diciembre de 2009
var variable2 = variable1;
```

```
// variable2 = 31 diciembre de 2010
variable2.setFullYear(2010, 11, 31);
// Ahora variable1 también es 31 diciembre de 2010
```

En el ejemplo anterior, se utiliza un tipo de dato de referencia que se verá más adelante, que se llama Date y que se utiliza para manejar fechas. Se crea una variable llamada **variable1** y se inicializa la fecha a 25 de diciembre de 2009. Al constructor del objeto Date se le pasa el año, el número del mes (siendo 0 = enero, 1 = febrero, ..., 11 = diciembre) y el día (al contrario que el mes, los días no empiezan en 0 sino en 1). A continuación, se asigna el valor de la **variable1** a otra variable llamada **variable2**.

Como Date es un tipo de referencia, la asignación se realiza por referencia. Por lo tanto, las dos variables quedan "unidas" y hacen referencia al mismo objeto, al mismo dato de tipo Date. De esta forma, si se modifica el valor de **variable2** (y se cambia su fecha a 31 de diciembre de 2010) el valor de **variable1** se verá automáticamente modificado.

CAPÍTULO 4: OPERADORES

Los operadores permiten manipular el valor de las variables, realizar operaciones matemáticas con sus valores y comparar diferentes variables. De esta forma, los operadores permiten a los programas realizar cálculos complejos y tomar decisiones lógicas en función de comparaciones y otros tipos de condiciones.

4.1 ASIGNACIÓN

El operador de asignación es el más utilizado y el más sencillo. Este operador se utiliza para guardar un valor específico en una variable. El símbolo utilizado es = (no confundir con el operador == que se verá más adelante):

```
var numero1 = 3;
```

A la izquierda del operador, siempre debe indicarse el nombre de una variable. A la derecha del operador, se pueden indicar variables, valores, condiciones lógicas, etc.:

```
var numero1 = 3;
var numero2 = 4;

5 = numero1; /* Error, la asignación siempre se realiza a una
                variable, por lo que en la izquierda no se puede
                indicar un número */

// Ahora, la variable numero1 vale 5
numero1 = 5;

// Ahora, la variable numero1 vale 4
numero1 = numero2;
```

4.2 ARITMÉTICOS

JavaScript permite realizar manipulaciones matemáticas sobre el valor de las variables numéricas. Los operadores definidos son: **suma (+), resta (-), multiplicación (*), división (/) y módulo (%).** Ejemplo:

```
var numero1 = 10;
var numero2 = 5;
resultado = numero1 / numero2;  // resultado = 2
resultado = 3 + numero1;        // resultado = 13
resultado = numero2 - 4;        // resultado = 1
resultado = numero1 * numero 2; // resultado = 50
resultado = numero1 % numero2;  // resultado = 0
numero1 = 9;
numero2 = 5;
```

```
resultado = numero1 % numero2;  // resultado = 4
```

Los operadores matemáticos también se pueden combinar con el operador de asignación para abreviar su notación:

```
var numero1 = 5;
numero1 += 3;   // numero1 = numero1 + 3 = 8
numero1 -= 1;   // numero1 = numero1 - 1 = 4
numero1 *= 2;   // numero1 = numero1 * 2 = 10
numero1 /= 5;   // numero1 = numero1 / 5 = 1
numero1 %= 4;   // numero1 = numero1 % 4 = 1
```

Existen dos operadores especiales que solamente son válidos para las variables numéricas y se utilizan para incrementar o decrementar en una unidad el valor de una variable.

Ejemplo:

```
var numero = 5;
++numero;
document.write(numero);  // numero = 6
```

El operador de incremento se indica mediante el prefijo ++ en el nombre de la variable. El resultado es que el valor de esa variable se incrementa en una unidad. Por tanto, el anterior ejemplo es equivalente a:

```
var numero = 5;
numero = numero + 1;
document.write(numero);  // numero = 6
```

De forma equivalente, el operador decremento (indicado como un prefijo -- en el nombre de la variable) se utiliza para decrementar el valor de la variable:

```
var numero = 5;
--numero;
document.write(numero);  // numero = 4
```

El anterior ejemplo es equivalente a:

```
var numero = 5;
numero = numero - 1;
document.write(numero);  // numero = 4
```

Los operadores de incremento y decremento no solamente se pueden indicar como prefijo del nombre de la variable, sino que también es posible utilizarlos como sufijo. En este caso, su comportamiento es diferente. Ejemplo:

```
var numero = 5;
numero++;
document.write(numero);  // numero = 6
```

El resultado de ejecutar el script anterior es el mismo que cuando se utiliza el operador ++numero, por lo que puede parecer que es equivalente indicar el operador ++ delante o detrás del identificador de la variable. Sin embargo, el siguiente ejemplo muestra sus diferencias:

```
var numero1 = 5;
var numero2 = 2;
numero3 = numero1++ + numero2;
// numero3 = 7, numero1 = 6
var numero1 = 5;
var numero2 = 2;
numero3 = ++numero1 + numero2;
// numero3 = 8, numero1 = 6
```

Si el operador ++ se indica como prefijo del identificador de la variable, su valor se incrementa antes de realizar cualquier otra operación. Si el operador ++ se indica como sufijo del identificador de la variable, su valor se incrementa después de ejecutar la sentencia en la que aparece.

Por tanto, en la instrucción numero3 = numero1++ + numero2; el valor de numero1 se incrementa después de realizar la operación (primero se suma y numero3 vale 7, después se incrementa el valor de numero1 y vale 6). Sin embargo, en la instrucción numero3 = ++numero1 + numero2; en primer lugar se incrementa el valor de numero1 y después se realiza la suma (primero se incrementa numero1 y vale 6, después se realiza la suma y numero3 vale 8).

4.3 LÓGICOS

Los operadores lógicos son imprescindibles para realizar aplicaciones complejas, ya que se utilizan para tomar decisiones sobre las instrucciones que debería ejecutar el programa en función de ciertas condiciones. El resultado de cualquier operación que utilice operadores lógicos siempre es un valor lógico o booleano.

4.3.1 NEGACIÓN

Uno de los operadores lógicos más utilizados es el de la negación. Se utiliza para obtener el valor contrario al valor de la variable:

```
var visible = true;
document.write(!visible);  // Muestra "false" y no "true"
```

La negación lógica se obtiene prefijando el símbolo **!** al identificador de la variable. El funcionamiento de este operador se resume en la siguiente tabla:

Variable	!variable
True	False
False	True

Si la variable original es de tipo *booleano*, es muy sencillo obtener su negación. Sin embargo, ¿qué sucede cuando la variable es un número o una cadena de texto? Para obtener la negación en este tipo de variables, se realiza en primer lugar su conversión a un valor booleano:

- Si la variable contiene un número, se transforma en false si vale 0 y en true para cualquier otro número (positivo o negativo, decimal o entero).
- Si la variable contiene una cadena de texto, se transforma en false si la cadena es vacía ("") y en true en cualquier otro caso.

```
var cantidad = 0;
vacio = !cantidad;  // vacio = true
cantidad = 2;
vacio = !cantidad;  // vacio = false
var mensaje = "";
mensajeVacio = !mensaje;  // mensajeVacio = true
mensaje = "Bienvenido";
mensajeVacio = !mensaje;  // mensajeVacio = false
```

4.3.2 AND

La operación lógica AND obtiene su resultado combinando dos valores booleanos. El operador se indica mediante el símbolo **&&** y su resultado solamente es true si los dos operandos son true:

variable1	variable2	variable1 && variable2
true	True	True

variable1	variable2	variable1 && variable2
true	False	False
false	True	False
false	False	False

```
var valor1 = true;
var valor2 = false;
resultado = valor1 && valor2; // resultado = false
valor1 = true;
valor2 = true;
resultado = valor1 && valor2; // resultado = true
```

4.3.3 OR

La operación lógica OR también combina dos valores booleanos. El operador se indica mediante el símbolo || (*código ascii 124* |) y su resultado es true si alguno de los dos operandos es true:

variable1	variable2	variable1 || variable2
true	True	True
true	False	False
false	True	True
false	False	False

```
var valor1 = true;
var valor2 = false;
resultado = valor1 || valor2; // resultado = true
valor1 = false;
```

```
valor2 = false;
resultado = valor1 || valor2; // resultado = false
```

4.4 RELACIONALES

Los operadores relacionales definidos por JavaScript son idénticos a los que definen las matemáticas: **mayor que (>), menor que (<), mayor o igual (>=), menor o igual (<=), igual que (==) y distinto de (!=).**
Los operadores que relacionan variables son imprescindibles para realizar cualquier aplicación compleja. El resultado de todos estos operadores siempre es un valor booleano:

```
var numero1 = 3;
var numero2 = 5;
resultado = numero1 > numero2;  // resultado = false
resultado = numero1 < numero2;  // resultado = true
numero1 = 5;
numero2 = 5;
resultado = numero1 >= numero2; // resultado = true
resultado = numero1 <= numero2; // resultado = true
resultado = numero1 == numero2; // resultado = true
resultado = numero1 != numero2; // resultado = false
```

Se debe tener especial cuidado con el operador de igualdad (==), ya que es el origen de la mayoría de errores de programación, incluso para los usuarios que ya tienen cierta experiencia desarrollando scripts. El operador == se utiliza para comparar el valor de dos variables, por lo que es muy diferente del operador =, que se utiliza para asignar un valor a una variable:

```
// El operador "=" asigna valores
var numero1 = 5;
resultado = numero1 = 3;  // numero1 = 3 y resultado = 3
// El operador "==" compara variables
var numero1 = 5;
resultado = numero1 == 3; // numero1 = 5 y resultado = false
```

Los operadores relacionales también se pueden utilizar con variables de tipo cadena de texto:

```
var texto1 = "hola";
var texto2 = "hola";
var texto3 = "adios";
resultado = texto1 == texto3; // resultado = false
resultado = texto1 != texto2; // resultado = false
resultado = texto3 >= texto2; // resultado = false
```

Cuando se utilizan cadenas de texto, los operadores "mayor que" (>) y "menor que" (<) siguen un razonamiento no intuitivo: se compara letra a letra comenzando desde la izquierda hasta que se encuentre una diferencia entre las dos cadenas de texto. Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas (a es menor que b, b es menor que c, A es menor que a, etc.)

4.5 TYPEOF

El operador typeof se emplea para determinar el tipo de dato que almacena una variable. Su uso es muy sencillo, ya que sólo es necesario indicar el nombre de la variable cuyo tipo se quiere averiguar. Los posibles valores de retorno del operador son: undefined, boolean, number, string para cada uno de los tipos primitivos y object para los valores de referencia y también para los valores de tipo null. El operador typeof no distingue entre las variables declaradas pero no inicializadas y las variables que ni siquiera han sido declaradas:

```
var variable1;
// devuelve "undefined", aunque la variable1 ha sido declarada
typeof variable1;
// devuelve "undefined", la variable2 no ha sido declarada
typeof variable2;
```

CAPÍTULO 5: ESTRUCTURAS DE CONTROL

5.1 ESTRUCTURA IF...ELSE

La estructura más utilizada en JavaScript y en la mayoría de lenguajes de programación es la estructura if. Se emplea para tomar decisiones en función de una condición. Su definición formal es:

```
if(condicion) {
  ...
}
```

Si la condición se cumple (es decir, si su valor es true) se ejecutan todas las instrucciones que se encuentran dentro del bloque {...}. Si la condición no se cumple (es decir, si su valor es false) no se ejecuta ninguna instrucción contenida en {...} y el programa continúa ejecutando el resto de instrucciones del script.

Ejemplo:

```
var mostrarMensaje = true;
if(mostrarMensaje) {
  document.write("Hola Mundo");
}
```

En el ejemplo anterior, el mensaje se muestra al usuario ya que la variable mostrarMensaje tiene un valor de true y por tanto, el programa entra dentro del bloque de instrucciones del if.

El ejemplo se podría reescribir también como:

```
var mostrarMensaje = true;
if(mostrarMensaje == true) {
  document.write("Hola Mundo");
}
```

En este caso, la condición es una comparación entre el valor de la variable mostrarMensaje y el valor true. Como los dos valores coinciden, la igualdad se cumple y por tanto la condición es cierta, su valor es true y se ejecutan las instrucciones contenidas en ese bloque del if.

La comparación del ejemplo anterior suele ser el origen de muchos errores de programación, al confundir los operadores == y =. Las comparaciones siempre se realizan con el operador ==, ya que el operador = solamente asigna valores:

```
var mostrarMensaje = true;
// Se comparan los dos valores
if(mostrarMensaje == false) {
  ...
}

if(mostrarMensaje = false) {
  ... // Error - Se asigna el valor "false" a la variable
}
```

La condición que controla el if() puede combinar los diferentes operadores lógicos y relacionales mostrados anteriormente:

```
var mostrado = false;
if(!mostrado) {
  document.write("Es la primera vez que se muestra el mensaje");
}
```

Los operadores AND y OR permiten encadenar varias condiciones simples para construir condiciones complejas:

```
var mostrado = false;
var usuarioPermiteMensajes = true;

if(!mostrado && usuarioPermiteMensajes) {
  document.write("Es la primera vez que se muestra el mensaje");
}
```

La condición anterior está formada por una operación AND sobre dos variables. A su vez, a la primera variable se le aplica el operador de negación antes de realizar la operación AND. De esta forma, como el valor de **mostrado** es false, el valor **!mostrado** sería true. Como la variable **usuarioPermiteMensajes** vale true, el resultado de !mostrado && usuarioPermiteMensajes sería igual a true && true, por lo que el resultado final de la condición del if() sería true y por tanto, se ejecutan las instrucciones que se encuentran dentro del bloque del if().

En ocasiones, las decisiones que se deben realizar no son del tipo *"si se cumple la condición, hazlo; si no se cumple, no hagas nada"*. Normalmente las condiciones suelen ser del tipo *"si se cumple esta condición, hazlo; si no se cumple, haz esto otro"*.
Para este segundo tipo de decisiones, existe una variante de la estructura if llamada if...else. Su definición formal es la siguiente:

```
if(condicion) {
  ...
}
else {
  ...
}
```

Si la condición se cumple (es decir, si su valor es true) se ejecutan todas las instrucciones que se encuentran dentro del if(). Si la condición no se cumple (es decir, si su valor es false) se ejecutan todas las instrucciones contenidas en else { }. Ejemplo:

```
var edad = 18;
if(edad >= 18) {
  document.write("Eres mayor de edad");
} else {
  document.write("Todavía eres menor de edad");
}
```

Si el valor de la variable edad es mayor o igual que el valor numérico 18, la condición del if() se cumple y por tanto, se ejecutan sus instrucciones y se muestra el mensaje "Eres mayor de edad". Sin embargo, cuando el valor de la variable edad no es igual o mayor que 18, la condición del if() no se cumple, por lo que automáticamente se ejecutan todas las instrucciones del bloque else { }. En este caso, se mostraría el mensaje "Todavía eres menor de edad".

El siguiente ejemplo compara variables de tipo cadena de texto:

```
var nombre = "";
if(nombre == "") {
  document.write("Aún no nos has dicho tu nombre");
} else {
  document.write("Hemos guardado tu nombre");
}
```

La condición del if() anterior se construye mediante el operador ==, que es el que se emplea para comparar dos valores (no confundir con el operador = que se utiliza para asignar valores). En el ejemplo anterior, si la cadena de texto almacenada en la variable nombre es vacía (es decir, es igual a "") se muestra el mensaje definido en el if(). En otro caso, se muestra el mensaje definido en el bloque else { }.

La estructura if...else se puede encadenar para realizar varias comprobaciones seguidas:

```
if(edad < 12) {
  document.write("Todavía eres muy pequeño");
} else if(edad < 19) {
  document.write("Eres un adolescente");
} else if(edad < 35) {
  document.write("Aun sigues siendo joven");
} else {
  document.write("Piensa en cuidarte un poco más");
}
```

No es obligatorio que la combinación de estructuras if...else acabe con la instrucción else, ya que puede terminar con una instrucción de tipo else if().

5.2 ESTRUCTURA SWITCH

La estructura switch es muy útil cuando la condición que evaluamos puede tomar muchos valores. Si utilizásemos una sentencia if...else, tendríamos que repetir la condición para los distintos valores.

```
if(dia == 1) {
  document.write("Hoy es lunes.");
} else if(dia == 2) {
  document.write("Hoy es martes.");
} else if(dia == 3) {
  document.write("Hoy es miércoles.");
} else if(dia == 4) {
```

```
  document.write("Hoy es jueves.");
} else if(dia == 5) {
  document.write("Hoy es viernes.");
} else if(dia == 6) {
  document.write("Hoy es sábado.");
} else if(dia == 7) {
  document.write("Hoy es domingo.");
}
```

En este caso es más conveniente utilizar una estructura de control de tipo switch, ya que permite ahorrarnos trabajo y producir un código más limpio. Su definición formal es la siguiente:

```
switch(dia) {
  case 1: document.write("Hoy es lunes."); break;
  case 2: document.write("Hoy es martes."); break;
  case 3: document.write("Hoy es miércoles."); break;
  case 4: document.write("Hoy es jueves."); break;
  case 5: document.write("Hoy es viernes."); break;
  case 6: document.write("Hoy es sábado."); break;
  case 7: document.write("Hoy es domingo."); break;
  default: document.write("No es un dia valido");
}
```

El comportamiento por defecto de la estructura switch es seguir evaluando el resto de cláusulas, aun cuando una de ellas haya cumplido la condición. Para evitar ese comportamiento, es necesario utilizar la sentencia **break** en las cláusulas que deseemos.

La cláusula case no tiene por qué ser una constante, sino que puede ser una expresión al igual que en la estructura if.
Por ejemplo, si ingresamos un número entero positivo y queremos evaluar en que rango está, podemos utilizar la siguiente estructura **switch**:

```
  x = parseInt(prompt("Ingrese un nro. positivo:", ""));
  switch (true) {
    case x > 0 && x < 10: alert("Entre 0 y 10"); break;
    case x > 0 && x < 20: alert("Entre 10 y 20"); break;
    case x < 0 || x > 100: alert("Nro. Invalido"); break;
    default: alert("Desconocido");
  }
```

El truco consiste en poner la constante booleana true para que la condición del switch sea siempre verdadera y siempre se evalúen los distintos case, donde allí incluimos las condiciones.

5.3 ESTRUCTURA FOR

Hasta ahora hemos empleado estructuras SECUENCIALES y CONDICIONALES (bifurcación simple y múltiple). Existen otros tipos de estructuras tan importantes como las anteriores que son las estructuras REPETITIVAS.

Una estructura repetitiva permite ejecutar una instrucción o un conjunto de instrucciones varias veces.

Una ejecución repetitiva de sentencias se caracteriza por:
- La o las sentencias que se repiten.
- El test o prueba de condición en cada repetición, que motivará que se repitan o no las sentencias.

Dentro de las estructuras repetitivas distinguimos dos situaciones: con cantidad conocida de veces y con cantidad desconocida de veces.
La estructura FOR se emplea en aquellas situaciones en las cuales CONOCEMOS la cantidad de veces que queremos que se ejecute el bloque de instrucciones. Ejemplo: cargar 10 números, ingresar 5 notas de alumnos, etc. Conocemos de antemano la cantidad de veces que queremos que el bloque se repita.

Sintaxis:

```
for (<Inicialización> ; <Condición> ; <Incremento o Decremento>)
{
  <Instrucciones>
}
```

Esta estructura repetitiva tiene tres argumentos: variable de inicialización, condición y variable de incremento o decremento.
Por último, hay que decir que la ejecución de la sentencia break dentro de cualquier parte del bucle provoca la salida inmediata del mismo.
La idea del funcionamiento de un bucle `for` es la siguiente: *"mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del for. Además, después de cada repetición, actualiza el valor de las variables que se utilizan en la condición"*.

- La "inicialización" es la zona en la que se establece los valores iniciales de las variables que controlan la repetición.
- La "condición" es el único elemento que decide si continua o se detiene la repetición.
- La "actualización" es el nuevo valor que se asigna después de cada repetición a las variables que controlan la repetición.

```
var mensaje = "Hola, estoy dentro de un bucle";
for(var i = 0; i < 5; i++) {
  document.write(mensaje);
}
```

La parte de la inicialización del bucle consiste en:

```
var i = 0;
```

Por tanto, en primer lugar se crea la variable `i` y se le asigna el valor de `0`. Esta zona de inicialización solamente se tiene en consideración justo antes de comenzar a ejecutar el bucle. Las siguientes repeticiones no tienen en cuenta esta parte de inicialización.
La zona de condición del bucle es:

```
i < 5
```

Los bucles se siguen ejecutando mientras se cumplan las condiciones y se dejan de ejecutar

justo después de comprobar que la condición no se cumple. En este caso, mientras la variable i valga menos de 5 el bucle se ejecuta indefinidamente.
Como la variable i se ha inicializado a un valor de 0 y la condición para salir del bucle es que i sea menor que 5, si no se modifica el valor de i de alguna forma, el bucle se repetiría indefinidamente.

Por ese motivo, es imprescindible indicar la zona de actualización, en la que se modifica el valor de las variables que controlan el bucle:

```
i++
```

En este caso, el valor de la variable i se incrementa en una unidad después de cada repetición. La zona de actualización se ejecuta después de la ejecución de las instrucciones que incluye el for.

Así, durante la ejecución de la quinta repetición el valor de i será 4. Después de la quinta ejecución, se actualiza el valor de i, que ahora valdrá 5. Como la condición es que i sea menor que 5, la condición ya no se cumple y las instrucciones del for no se ejecutan una sexta vez.

Normalmente, la variable que controla los bucles for se llama i, ya que recuerda a la palabra índice y su nombre tan corto ahorra mucho tiempo y espacio.

Ejemplo: Mostrar por pantalla los números del 1 al 10.

```
<html>
<head>
</head>
<body>
<script>
  var f;
  for(f=1;f<=10;f++){
    document.write(f+" ");
  }
</script>
</body>
</html>
```

Inicialmente f se la inicializa con 1. Como la condición se verifica como verdadera se ejecuta el bloque del for (en este caso mostramos el contenido de la variable f y un espacio en blanco). Luego de ejecutar el bloque pasa al tercer argumento del for (en este caso con el operador ++ se incrementa en uno el contenido de la variable f, existe otro operador -- que decrementa en uno una variable), hubiera sido lo mismo poner f=f+1 pero este otro operador matemático nos simplifica las cosas.

Importante: Tener en cuenta que no lleva punto y coma al final de los tres argumentos del for. El disponer un punto y coma provoca un error lógico y no sintáctico, por lo que el navegador no avisará.

5.4 ESTRUCTURAS WHILE – DO WHILE

5.4.1 WHILE

Sintaxis del **while**: while (condicion) {
operaciones;
}

Funcionamiento del **while**: En primer lugar se verifica la condición, si la misma resulta verdadera se ejecutan las operaciones que indicamos entre las llaves que le siguen al while. En caso que la condición sea Falsa continúa con la instrucción siguiente al bloque de llaves. El bloque se repite MIENTRAS la condición sea Verdadera.

Importante: Si la condición siempre retorna verdadero estamos en presencia de un ciclo repetitivo infinito. Dicha situación es un error de programación, nunca finalizará el programa.

Ejemplo: Realizar un programa que imprima en pantalla los números del 1 al 100. Sin conocer las estructuras repetitivas podemos resolver el problema empleando una estructura secuencial. Inicializamos una variable con el valor 1, luego imprimimos la variable, incrementamos nuevamente la variable y así sucesivamente. Pero esta solución es muy larga. La mejor forma de resolver este problema es emplear una estructura repetitiva:

```
<html>
<head>
</head>
<body>
<script>
  var x;
  x=1;
  while (x<=100)
  {
    document.write(x);
    document.write('<br>');
    x=x+1;
  }
</script>
</body>
</html>
```

Para que se impriman los números, uno en cada línea, agregamos la marca HTML de
.
Es muy importante analizar este programa:
La primera operación inicializa la variable x en 1, seguidamente comienza la estructura repetitiva while y disponemos la siguiente condición (x <= 100), se lee MIENTRAS la variable x sea menor o igual a 100.
Al ejecutarse la condición, retorna VERDADERO, porque el contenido de x (1) es menor o igual a 100.
Al ser la condición verdadera se ejecuta el bloque de instrucciones que contiene la estructura while. El bloque de instrucciones contiene dos salidas al documento y una operación. Se imprime el contenido de x y seguidamente se incrementa la variable x en uno.

La operación x = x + 1 se lee como "en la variable x se guarda el contenido de x más 1". Es decir, si x contiene 1 luego de ejecutarse esta operación se almacenará en x un 2.

Al finalizar el bloque de instrucciones que contiene la estructura repetitiva, se verifica nuevamente la condición de la estructura repetitiva y se repite el proceso explicado anteriormente.
Mientras la condición retorne verdadero, se ejecuta el bloque de instrucciones; al retornar falso la verificación de la condición, se sale de la estructura repetitiva y continúa el algoritmo, en este caso, finaliza el programa.

Lo más difícil es la definición de la condición de la estructura while y qué bloque de instrucciones se va a repetir. Observar que si, por ejemplo, disponemos la condición x >=100 (si x es mayor o igual a 100) no provoca ningún error sintáctico pero estamos en presencia de un error lógico porque al evaluarse por primera vez la condición retorna falso y no se ejecuta el bloque de instrucciones que queríamos repetir 100 veces.

No existe una RECETA para definir una condición de una estructura repetitiva, sino que se logra con una práctica continua, solucionando problemas.

Una vez planteado el programa debemos verificar si el mismo es una solución válida al problema (en este caso se deben imprimir los números del 1 al 100 en la página), para ello podemos hacer un seguimiento del flujo del diagrama y los valores que toman las variables a lo largo de la ejecución:

x	
1	
2	
3	
4	
.	
.	
100	
101	*Cuando x vale 101 la condición de la estructura repetitiva retorna falso, en este caso finaliza el diagrama.*

La variable x recibe el nombre de CONTADOR. Un contador es un tipo especial de variable que se incrementa o decrementa con valores constantes durante la ejecución del programa. El contador x nos indica en cada momento la cantidad de valores impresos en la página.

Importante: Podemos observar que el bloque repetitivo puede no ejecutarse si la condición retorna falso la primera vez.
La variable x debe estar inicializada con algún valor antes que se ejecute la operación x = x + 1.

Probemos algunas modificaciones de este programa y veamos qué cambios se deberían hacer para:
1 - Imprimir los números del 1 al 500.
2 - Imprimir los números del 50 al 100.
3 - Imprimir los números del -50 al 0.
4 - Imprimir los números del 2 al 100 pero de 2 en 2 (2,4,6,8 ....100).

5.4.2 DO WHILE

La sentencia **do/while** es otra estructura repetitiva, la cual ejecuta al menos una vez su bloque repetitivo, a diferencia del **while** que puede no ejecutar el bloque.
Esta estructura repetitiva se utiliza cuando conocemos de antemano que por lo menos una vez se ejecutará el bloque repetitivo.
La condición de la estructura está abajo del bloque a repetir, a diferencia del while que está en la parte superior.
Finaliza la ejecución del bloque repetitivo cuando la condición retorna falso, es decir igual que el while.

Problema: Escribir un programa que solicite la carga de un número entre 0 y 999, y nos muestre un mensaje de cuántos dígitos tiene el mismo. Finalizar el programa cuando se cargue el valor 0.

```
<html>
<head> </head>
<body>
<script>
  var valor;
  do {
    do {
       valor=prompt('Ingrese un valor entre 1 y 999','<0 para salir>');
       valor=parseInt(valor);
    } while (valor<0 || valor>999);
    if (valor>0) {
              document.write('El valor ' + valor + ' tiene ');
              if (valor<10) {
                document.write('1 digito'+'<br>');
              } else {
                if (valor<100){
                     document.write('2 digitos'+'<br>');
                } else {
                     document.write('3 digitos'+'<br>');
                   }
                }
    }
  } while(valor!=0);
</script>
</body>
</html>
```

En este problema por lo menos se carga un valor. Si se carga un valor menor a 10 se trata de un número de una cifra, si es mayor a 10 pero menor a 100 se trata de un valor de dos dígitos, en caso contrario se trata de un valor de tres dígitos. Este bloque se repite mientras se ingresa en la variable 'valor' un número distinto a 0.

CAPÍTULO 6: FUNCIONES

6.1 Definición

En programación es muy frecuente que un determinado procedimiento de cálculo definido por un grupo de sentencias tenga que repetirse varias veces, ya sea en un mismo programa o en otros programas, lo cual implica que se tenga que escribir tantos grupos de aquellas sentencias como veces aparezca dicho proceso.
La herramienta más potente con que se cuenta para facilitar, reducir y dividir el trabajo en programación, es escribir aquellos grupos de sentencias una sola y única vez bajo la forma de una FUNCION.
Un programa es una cosa compleja de realizar y por lo tanto es importante que esté bien ESTRUCTURADO y también que sea inteligible para las personas. Si un grupo de sentencias realiza una tarea bien definida, entonces puede estar justificado el aislar estas sentencias formando una función, aunque resulte que sólo se le llame o use una vez.
Hasta ahora hemos visto como resolver un problema planteando un único algoritmo.
Con funciones podemos segmentar un programa en varias partes.
Frente a un problema, planteamos un algoritmo, éste puede constar de pequeños algoritmos.
Una función es un conjunto de instrucciones que resuelven una parte del problema y que puede ser utilizado (llamado) desde diferentes partes de un programa.
Consta de un nombre y parámetros. Con el nombre llamamos a la función, es decir, hacemos referencia a la misma. Los parámetros son valores que se envían y son indispensables para la resolución del mismo. La función realizará alguna operación con los parámetros que le enviamos.
Podemos cargar una variable, consultarla, modificarla, imprimirla, etc.

Incluso los programas más sencillos tienen la necesidad de fragmentarse. Las funciones son los únicos tipos de subprogramas que acepta JavaScript. Tienen la siguiente estructura:

```
function <nombre de función>(argumento1, argumento2, ..., argumento n)
{
  <código de la función>
}
```

Debemos buscar un nombre de función que nos indique cuál es su objetivo (Si la función recibe un string y lo centra, tal vez deberíamos llamarla centrarTitulo). Veremos que una función puede variar bastante en su estructura, puede tener o no parámetros, retornar un valor, etc.

Ejemplo: Mostrar un mensaje que se repita 3 veces en la página con el siguiente texto:

```
'Cuidado'
'Ingrese su documento correctamente'
'Cuidado'
'Ingrese su documento correctamente'
'Cuidado'
'Ingrese su documento correctamente'
```

La solución sin emplear funciones es:

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  document.write("Cuidado<br>");
  document.write("Ingrese su documento correctamente<br>");
  document.write("Cuidado<br>");
  document.write("Ingrese su documento correctamente<br>");
  document.write("Cuidado<br>");
  document.write("Ingrese su documento correctamente<br>");
</script>
</body>
</html>
Empleando una función:
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function mostrarMensaje()
  {
    document.write("Cuidado<br>");
    document.write("Ingrese su documento correctamente<br>");
  }
  mostrarMensaje();
  mostrarMensaje();
  mostrarMensaje();
</script>
</body>
</html>
```

Recordemos que JavaScript es sensible a mayúsculas y minúsculas. Si fijamos como nombre a la función mostrarTitulo (es decir la segunda palabra con mayúscula) debemos respetar este nombre cuando la llamemos a dicha función.
Es importante notar que para que una función se ejecute debemos llamarla desde fuera por su nombre (en este ejemplo: mostrarMensaje()).
Cada vez que se llama una función se ejecutan todas las líneas contenidas en la misma.
Si no se llama a la función, las instrucciones de la misma nunca se ejecutarán.
A una función la podemos llamar tantas veces como necesitemos.
Las funciones nos ahorran escribir código que se repite con frecuencia y permite que nuestro programa sea más entendible.

6.2 Funciones con parámetros

Explicaremos con un ejemplo, una función que tiene datos de entrada.

Ejemplo: Confeccionar una función que reciba dos números y muestre en la página los valores comprendidos entre ellos de uno en uno. Cargar por teclado esos dos valores.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function mostrarComprendidos(x1,x2)
  {
    var inicio;
    for(inicio=x1 ; inicio<=x2 ; inicio++)
    {
      document.write(inicio+' ');
    }
  }
  var valor1,valor2;
  valor1=prompt('Ingrese valor inferior:','');
  valor1=parseInt(valor1);
  valor2=prompt('Ingrese valor superior:','');
  valor2=parseInt(valor2);
  mostrarComprendidos(valor1,valor2);
</script>
</body>
</html>
```

El programa de JavaScript empieza a ejecutarse donde definimos las variables valor1 y valor2 y no donde se define la función. Luego de cargar los dos valores por teclado se llama a la función mostrarComprendidos y le enviamos las variables valor1 y valor2. Los parámetros x1 y x2 reciben los contenidos de las variables valor1 y valor2.
Es importante notar que a la función la podemos llamar la cantidad de veces que la necesitemos. Los nombres de los parámetros, en este caso se llaman x1 y x2, no necesariamente se deben llamar igual que las variables que le pasamos cuando la llamamos a la función, en este caso le pasamos los valores valor1 y valor2.

6.3 Funciones que retornan un valor

Son comunes los casos donde una función, luego de hacer un proceso, retorne un valor.
Ejemplo 1: Confeccionar una función que reciba un valor entero comprendido entre 1 y 5. Luego retornar en castellano el valor recibido.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function convertirCastellano(x)
```

```
  {
    if (x==1)
      return "uno";
    else
      if (x==2)
        return "dos";
      else
        if (x==3)
          return "tres";
        else
          if (x==4)
            return "cuatro";
          else
            if (x==5)
              return "cinco";
            else
              return "valor incorrecto";
  }
  var valor;
  valor=prompt("Ingrese un valor entre 1 y 5","");
  valor=parseInt(valor);
  var r;
  r=convertirCastellano(valor);
  document.write(r);
</script>
</body>
</html>
```

Podemos ver que el valor retornado por una función lo indicamos por medio de la palabra clave return. Cuando se llama a la función, debemos asignar el nombre de la función a una variable, ya que la misma retorna un valor.
Una función puede tener varios parámetros, pero sólo puede retornar un único valor.
La estructura condicional if de este ejemplo puede ser remplazada por la instrucción switch, la función queda codificada de la siguiente manera:

```
function convertirCastellano(x)
{
  switch (x)
  {
    case 1:return "uno";
    case 2:return "dos";
    case 3:return "tres";
    case 4:return "cuatro";
    case 5:return "cinco";
    default:return "valor incorrecto";
  }
```

```
}
```

Esta es una forma más elegante que una serie de if anidados. La instrucción switch analiza el contenido de la variable x con respecto al valor de cada caso. En la situación de ser igual, ejecuta el bloque seguido de los 2 puntos hasta que encuentra la instrucción return o break.

Ejemplo 2: Confeccionar una función que reciba una fecha con el formato de día, mes y año y retorne un string con un formato similar a: "Hoy es 10 de junio de 2013".

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function formatearFecha(dia,mes,anio)
  {
    var s='Hoy es '+dia+' de ';
    switch (mes) {
    case 1:s=s+'enero ';
           break;
    case 2:s=s+'febrero ';
           break;
    case 3:s=s+'marzo ';
           break;
    case 4:s=s+'abril ';
           break;
    case 5:s=s+'mayo ';
           break;
    case 6:s=s+'junio ';
           break;
    case 7:s=s+'julio ';
           break;
    case 8:s=s+'agosto ';
           break;
    case 9:s=s+'septiembre ';
           break;
    case 10:s=s+'octubre ';
           break;
    case 11:s=s+'noviembre ';
           break;
    case 12:s=s+'diciembre ';
           break;
    } //fin del switch
    s=s+'de '+anio;
    return s;
  }
```

```
  document.write(formatearFecha(11,6,2013));
</script>
</body>
</html>
```

Analicemos un poco la función formatearFecha. Llegan tres parámetros con el día, mes y año. Definimos e inicializamos una variable con:

```
var s='Hoy es '+dia+' de ';
```

Luego le concatenamos o sumamos el mes:

```
s=s+'enero ';
```

Esto, si el parámetro mes tiene un uno. Observemos como acumulamos lo que tiene 's' más el string 'enero '. En caso de hacer s='enero ' perderíamos el valor previo que tenía la variable s. Por último concatenamos el año:

```
s=s+'de '+año;
```

Cuando se llama a la función directamente, al valor devuelto se lo enviamos a la función write del objeto document. Esto último lo podemos hacer en dos pasos:

```
var fec= formatearFecha(11,6,2013);
document.write(fec);
```

Guardamos en la variable 'fec' el string devuelto por la función.

Archivo JavaScript externo (*.js)

El lenguaje JavaScript permite agrupar funciones y disponerlas en un archivo separado a la página HTML.
Esto trae muchos beneficios:
- Reutilización de funciones en muchos archivos. No tenemos que copiar y pegar sucesivamente las funciones en las páginas en las que necesitamos.
- Facilita el mantenimiento de las funciones al encontrarse en archivos separados.
- Nos obliga a ser más ordenados.

La mecánica para implementar estos archivos externos en JavaScript es:

1 - Crear un archivo con extensión *.js y tipear las funciones en la misma:

```
function retornarFecha(){
  var fecha
  fecha=new Date();
  var cadena=fecha.getDate()+'/'+(fecha.getMonth()+1)+'/'+fecha.getYear();
  return cadena;
}

function retornarHora(){
  var fecha
  fecha=new Date();
```

```
  var cadena=fecha.getHours()+':'+fecha.getMinutes()+':'+fecha.getSeconds();
  return cadena;
}
```

2 - Creamos un archivo html que utilizará las funciones contenidas en el archivo *.js:

```
<html>
<head>
<title>Problema</title>
<script src="fechas.js">
</script>
</head>
<body>
<script>
document.write('La fecha de hoy es '+retornarFecha());
document.write('<br>');
document.write('La hora es '+retornarHora());
</script>
</body>
</html>
```

Es decir debemos disponer el siguiente código para importar el archivo *.js:

```
<script src="fechas.js">
</script>
```

Mediante la propiedad **src** indicamos el nombre del archivo a importar.
Luego, podemos llamar dentro de la página HTML a las funciones que contiene el archivo externo *.js .
En nuestro ejemplo llamamos a las funciones retornarFecha() y retornarHora().
Como podemos ver, el archivo html queda mucho más limpio.

CAPÍTULO 7: FUNCIONES PREDEFINIDAS

7.1 Introducción a la Programación Orientada a Objetos.

Un objeto es una estructura que contiene tanto las variables (llamadas propiedades) como las funciones que manipulan dichas variables (llamadas métodos). A partir de esta estructura se ha creado un nuevo modelo de programación (la programación orientada a objetos) que atribuye a los mismos propiedades como herencia o polimorfismo. Como veremos, JavaScript simplifica en algo este modelo y hace una programación híbrida entre la programación estructurada y la programación orientada a objetos.
El modelo de la programación orientada a objetos normal y corriente separa los mismos en dos: clases e instancias (objetos). Las primeras son entes más abstractos que definen un conjunto determinado de objetos. Las segundas son miembros de una clase, poseyendo las mismas propiedades que la clase a la cual pertenecen.

7.1.1 Propiedades y métodos.

Para acceder a los métodos y propiedades de un objeto debemos utilizar la siguiente sintaxis:
objeto.propiedad
objeto.metodo(parametros)

7.1.2 Conceptos Básicos.

Objetos

Son todas las cosas con identidad propia. Se relacionan entre si. Poseen características (atributos) y tienen responsabilidades (funciones, métodos) que deben cumplir. Son ejemplares (instancias) de una clase y conocen a la clase a la cual pertenecen.

Atributos o propiedades

Son las características, cualidades distintivas de cada objeto. Deben ser mínimos para poder realizar todas las operaciones que requiere la aplicación.

Ejemplos de objetos del mundo real:
- Casa:
  atributos: tamaño, precio, cantidad de habitaciones, etc.;
  responsabilidades: comodidad, seguridad, etc.
- Mesa:
  atributos: altura, largo, ancho, etc.;
  responsabilidades: contener elementos.
- Ventana:
  atributos: tamaño, color, etc.;
  responsabilidades: abrirse, cerrarse, etc.

Ejemplos de objetos del mundo de la programación:
- Ventana:
  atributos: tamaño, color, etc.;
  responsabilidades: mostrar título, achicarse, etc.

Responsabilidades o Métodos.

Son las responsabilidades que debe cumplir la clase. El objetivo de un método es ejecutar las actividades que tiene encomendada la clase.
Es un algoritmo (conjunto de operaciones) que se ejecuta en respuesta a un mensaje; respuestas a mensajes para satisfacer peticiones.
Un método consiste en el nombre de la operación y sus argumentos. El nombre del método identifica una operación que se ejecuta.
Un método está determinado por la clase del objeto receptor, todos los objetos de una clase usan el mismo método en respuesta a mensajes similares.
La interpretación de un mensaje (selección del método ejecutado) depende del receptor y puede variar con distintos receptores, es decir, puede variar de una clase a otra.

Clases

Una clase es un molde para objetos que poseen las mismas características (que pueden recibir los mismos mensajes y responden de la misma manera).
Una clase es una representación de una idea o concepto. Unidad que encapsula códigos y datos para los métodos (operaciones).
Todos los ejemplares de una clase se comportan de forma similar (invocan el mismo método) en respuesta a mensajes similares.
La clase a la cual pertenece un objeto determina el comportamiento del objeto.
Una clase tiene encomendadas actividades que ejecutan los métodos.
Las clases están definidas por:

- Atributos (Propiedades),
- Comportamiento (operaciones o métodos) y
- Relaciones con otros objetos.

Una aplicación es un conjunto de objetos de determinadas clases.

JavaScript incorpora una serie de herramientas y utilidades (llamadas funciones y propiedades, como se verá más adelante) para el manejo de las variables. De esta forma, muchas de las operaciones básicas con las variables, se pueden realizar directamente con las utilidades que ofrece JavaScript.

7.2 Funciones de Cadena (Clase String)

Un string consiste en uno o más caracteres encerrados entre comillas simples o dobles.

7.2.1 Concatenación de cadenas (+)

JavaScript permite concatenar cadenas utilizando el operador +.
El siguiente fragmento de código concatena tres cadenas para producir su salida:

```
var final='La entrada tiene ' + contador + ' caracteres.';
```

Dos de las cadenas concatenadas son cadenas literales. La del medio es un entero que automáticamente se convierte a cadena y luego se concatena con las otras.

7.2.2 Propiedad length

Retorna la cantidad de caracteres de un objeto String.

```
var nom='Juan';
document.write(nom.length); //Resultado 4
```

7.2.3 Métodos

- **charAt(pos)**

Retorna el carácter del índice especificado. Comienzan a numerarse de la posición cero.

```
var nombre='juan';
var caracterPrimero=nombre.charAt(0);
```

- **substring (posinicial, posfinal)**

Retorna un String extraído de otro, desde el carácter 'posinicial' hasta el 'posfinal'-1:

```
cadena3=cadena1.substring(2,5);
```

En este ejemplo, "cadena3" contendrá los caracteres 2, 3, 4 sin incluir el 5 de cadena1 (Cuidado que comienza en cero).

- **indexOf (subCadena)**

Devuelve la posición de la subcadena dentro de la cadena, o -1 en caso de no estar.
Tener en cuenta que puede retornar 0 si la subcadena coincide desde el primer carácter.

```
var nombre='Rodriguez Pablo';
var pos=nombre.indexOf('Pablo');
if (pos!=-1) {document.write ('Está el nombre Pablo en la variable nombre');}
```

- **lastIndexOf(caracter),**

Calcula la última posición en la que se encuentra el caracter indicado dentro de la cadena de texto. Si la cadena no contiene el caracter, la función devuelve el valor -1:

```
var mensaje = "Hola";
var posicion = mensaje.lastIndexOf('a'); // posicion = 3
posicion = mensaje.lastIndexOf('b');     // posicion = -1
```

La función lastIndexOf() comienza su búsqueda desde el final de la cadena hacia el principio, aunque la posición devuelta es la correcta empezando a contar desde el principio de la palabra.

- **toUpperCase()**

Convierte todos los caracteres del String que invoca el método a mayúsculas:

```
cadena1=cadena1.toUpperCase();
```

Luego de esto, cadena1 tiene todos los caracteres convertidos a mayúsculas.

- **toLowerCase()**

Convierte todos los caracteres del String que invoca el método a minúsculas:

```
cadena1=cadena1.toLowerCase();
```

Luego de esto, cadena1 tiene todos los caracteres convertidos a minúsculas.

- **split(separador)**

Convierte una cadena de texto en un array de cadenas de texto. La función parte la cadena de texto determinando sus trozos a partir del caracter separador indicado:

```
var mensaje = "Hola Mundo, soy una cadena de texto!";
var palabras = mensaje.split(" ");
// palabras = ["Hola", "Mundo,", "soy", "una", "cadena", "de", "texto!"];
```

Con esta función se pueden extraer fácilmente las letras que forman una palabra:

```
var palabra = "Hola";
```

```
var letras = palabra.split(""); // letras = ["H", "o", "l", "a"]
```

Ejemplo: Cargar un string por teclado y luego llamar a los distintos métodos de la clase String y la propiedad length.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  var cadena=prompt('Ingrese una cadena:','');
  document.write('La cadena ingresada es:'+cadena);
  document.write('<br>');
  document.write('La cantidad de caracteres son:'+cadena.length);
  document.write('<br>');
  document.write('El primer carácter es:'+cadena.charAt(0));
  document.write('<br>');
  document.write('Los primeros 3 caracteres
son:'+cadena.substring(0,3));
  document.write('<br>');
  if (cadena.indexOf('hola')!=-1)
    document.write('Se ingresó la subcadena hola');
  else
    document.write('No se ingresó la subcadena hola');
  document.write('<br>');
  document.write('La cadena convertida a mayúsculas
es:'+cadena.toUpperCase());
  document.write('<br>');
  document.write('La cadena convertida a minúsculas
es:'+cadena.toLowerCase());
  document.write('<br>');
</script>

</body>
</html>
```

7.3 Funciones útiles para números

A continuación se muestran algunas de las funciones y propiedades más útiles para el manejo de números.

NaN, (del inglés, "Not a Number") JavaScript emplea el valor NaN para indicar un valor num érico no definido (por ejemplo, la división 0/0).

```
var numero1 = 0;
```

```
var numero2 = 0;
document.write (numero1/numero2); // se muestra el valor NaN
```

isNaN(), permite proteger a la aplicación de posibles valores numéricos no definidos

```
var numero1 = 0;
var numero2 = 0;
if(isNaN(numero1/numero2)) {
  document.write("La división no está definida para los números
indicados");
} else {
  document.write("La división es igual a => " + numero1/numero2);
}
```

Infinity, hace referencia a un valor numérico infinito y positivo (también existe el valor –infinity para los infinitos negativos)

```
var numero1 = 10;
var numero2 = 0;
document.write (numero1/numero2); // se muestra el valor Infinity
```

toFixed(digitos), devuelve el número original con tantos decimales como los indicados por el parámetro digitos y realiza los redondeos necesarios. Se trata de una función muy útil por ejemplo para mostrar precios.

```
var numero1 = 4564.34567;
numero1.toFixed(2); // 4564.35
numero1.toFixed(6); // 4564.345670
numero1.toFixed(); // 4564
```

7.3.1 Clase Math

Esta clase es un contenedor que tiene diversas constantes (como Math.E y Math.PI) y los siguientes métodos matemáticos:

Método	Descripción	Expresión de ejemplo	Resultado del ejemplo
abs	Valor absoluto	Math.abs(-2)	2
sin, cos, tan	Funciones trigonométricas, reciben el argumento en radianes	Math.cos(Math.PI)	-1

asin, acos, atan	Funciones trigonométricas inversas	Math.asin(1)	1.57
exp, log	Exponenciación y logaritmo, base E	Math.log(Math.E)	1
ceil	Devuelve el entero más pequeño mayor o igual al argumento	Math.ceil(-2.7)	-2
floor	Devuelve el entero más grande menor o igual al argumento	Math.floor(-2.7)	-3
round	Devuelve el entero más cercano o igual al argumento	Math.round(-2.7)	-3
min, max	Devuelve el menor (o mayor) de sus dos argumentos	Math.min(2,4)	2
pow	Exponenciación, siendo el primer argumento la base y el segundo el exponente	Math.pow(2,3)	8
sqrt	Raíz cuadrada	Math.sqrt(25)	5
random	Genera un valor aleatorio comprendido entre 0 y 1.	Math.random()	Ej. 0.7345

Ejemplo: Confeccionar un programa que permita cargar un valor comprendido entre 1 y 10. Luego generar un valor aleatorio entre 1 y 10, mostrar un mensaje con el número sorteado e indicar si ganó o perdió:

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  var selec=prompt('Ingrese un valor entre 1 y 10','');
  selec=parseInt(selec);
  var num=parseInt(Math.random()*10)+1;
  if (num==selec)
    document.write('Ganó el número que se sorteó es el '+ num);
  else
```

```
    document.write('Lo siento se sorteó el valor '+num+' y usted
eligió el '+selec);
</script>
</body>
</html>
```

Para generar un valor aleatorio comprendido entre 1 y 10 debemos plantear lo siguiente:

```
  var num=parseInt(Math.random()*10)+1;
```

Al multiplicar Math.random() por 10, nos genera un valor aleatorio comprendido entre un valor mayor a 0 y menor a 10, luego, con la función parseInt, obtenemos sólo la parte entera. Finalmente sumamos uno.

El valor que cargó el operador se encuentra en:

```
  var selec=prompt('Ingrese un valor entre 1 y 10','');
```

Con un simple if validamos si coinciden los valores (el generado y el ingresado por teclado)

7.4 Funciones para manejo de Fechas (Clase Date).

JavaScript dispone de varias clases predefinidas para acceder a muchas de las funciones normales de cualquier lenguaje, como puede ser el manejo de vectores o el de fechas.

La clase Date nos permitirá manejar fechas y horas. Se invoca así:

```
fecha = new Date(); //creación de un objeto de la clase Date
fecha = new Date(año, mes, dia);
fecha = new Date(año, mes, dia, hora, minuto, segundo);
```

Si no utilizamos parámetros, el objeto fecha contendrá la fecha y hora actuales, obtenidas del reloj de nuestra computadora. En caso contrario hay que tener en cuenta que los meses comienzan por cero. Así, por ejemplo:

```
navidad06 = new Date(2006, 11, 25)
```

El objeto Date dispone, entre otros, de los siguientes métodos (suponemos que **fecha** es un objeto de tipo date):

```
  fecha.getYear()
  fecha.setYear(año)
```

Obtiene y coloca, respectivamente, el año de la fecha.
Éste se devuelve como número de 4 dígitos excepto en el caso en que esté entre 1900 y 1999, en cuyo caso devolverá las dos últimas cifras.

```
  fecha.getFullYear()
  fecha.setFullYear(año)
```

Realizan la misma función que los anteriores, pero sin tanta complicación, ya que siempre

devuelven números con todos sus dígitos.

```
fecha.getMonth()
fecha.setMonth(mes)
fecha.getDate()
fecha.setDate(dia)
fecha.getHours()
fecha.setHours(horas)
fecha.getMinutes()
fecha.setMinutes(minutos)
fecha.getSeconds()
fecha.setSeconds(segundos)
```

Obtienen y colocan, respectivamente, el mes, día, hora, minuto y segundo de la fecha.

```
fecha.getDay()
```

Devuelve el día de la semana de la fecha en forma de número que va del 0 (domingo) al 6 (sábado)

Ejemplo: Mostrar en una página la fecha y la hora actual.

```
<HTML>
<HEAD>
<SCRIPT type="text/javascript">
  function mostrarFechaHora()
  {
    var fecha
    fecha=new Date();
    document.write('Hoy es ');
    document.write(fecha.getDate()+'/');
    document.write((fecha.getMonth()+1)+'/');
    document.write(fecha.getFullYear());
    document.write('<br>');
    document.write('Es la hora ');
    document.write(fecha.getHours()+':');
    document.write(fecha.getMinutes()+':');
    document.write(fecha.getSeconds());
  }
//Llamada a la función
  mostrarFechaHora();
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

En este problema hemos creado un objeto de la clase Date. Luego llamamos una serie de métodos que nos retornan datos sobre la fecha y hora actual del equipo de computación donde se está ejecutando el navegador.

Es bueno notar que para llamar a los métodos disponemos:
<nombre de objeto>.<nombre de método>(parámetros)

7.5 Arrays

Un vector o Array es una estructura de datos que permite almacenar un CONJUNTO de datos. Con un único nombre se define un arreglo y por medio de un subíndice hacemos referencia a cada elemento del mismo (componente).

Un array es una asignación lineal de memoria donde los elementos son accedidos a través de índices numéricos, siendo además una estructura de datos muy rápida. Desafortunadamente, JavaScript no utiliza este tipo de arrays. En su lugar, JavaScript ofrece un objeto que dispone de características que le hacen parecer un array. Internamente, convierte los índices del array en strings que son utilizados como nombres de propiedades, haciéndolo sensiblemente más lento que un array.

Un array en JavaScript puede almacenar en sus componentes elementos de datos distintos y su tamaño puede crecer a lo largo de la ejecución del programa.

7.5.1 REPRESENTACIÓN DE UN ARRAY

Tenemos muchas formas de inicializar un array en Javascript según nuestra situación particular, veamos con ejemplos diferentes formas:
Creación de un array sin elementos:

```
var vector1=new Array();
```

Otra sintaxis para crear un array sin elementos:

```
var vector2=[];
```

Creación de un array indicando la cantidad de componentes iniciales que podrá almacenar:

```
var vector3=new Array(5);
```

Creación e inicialización llamando al constructor Array y pasando como parámetros los valores a almacenar en las componentes:

```
var vector4=new Array(1,70,'juan');
```

Creación e inicialización de un array utilizando los corchetes:

```
var vector5=[1,70,'juan'];
```

7.5.2 PROPIEDAD LENGTH

Todo array tiene una propiedad length. A diferencia de otros lenguajes, la longitud del array no es fija, y podemos añadir elementos de manera dinámica. Esto hace que la propiedad length varíe, y tenga en cuenta los nuevos elementos. La propiedad length hace referencia al mayor índice presente en el array, más uno. Esto es:

```
var myArray = [];
myArray.length              // 0
myArray[1000000] = true;
myArray.length              // 1000001
// myArray contiene un elemento!
```

La propiedad length puede indicarse de manera explícita. Aumentando su valor, no vamos a reservar más espacio para el array, pero si disminuimos su valor, haciendo que sea menor que el número de elementos del array, eliminará los elementos cuyo índice sea mayor que el nuevo length:

```
numbers.length = 3; // numbers es ['zero', 'one', 'two']
```

Ejemplo 1: Crear un array para almacenar los cinco sueldos de operarios y luego mostrar el total de gastos en sueldos (cada actividad en una función).

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function cargar(sueldos)
  {
    var f;
    for(f=0;f<sueldos.length;f++)
    {
      var v;
      v=prompt('Ingrese sueldo:','');
      sueldos[f]=parseInt(v);
    }
  }
  function calcularGastos(sueldos)
  {
    var total=0;
    var f;
    for(f=0;f<sueldos.length;f++)
    {
      total=total+sueldos[f];
    }
    document.write('Listado de sueldos<br>');
```

```
    for(f=0;f<sueldos.length;f++)
    {
      document.write(sueldos[f]+'<br>');
    }
    document.write('Total de gastos en sueldos:'+total);
  }
  var sueldos;
  sueldos=new Array(5);
  cargar(sueldos);
  calcularGastos(sueldos);
</script>
</body>
</html>
```

Recordemos que el programa comienza a ejecutarse a partir de las líneas que se encuentran fuera de la funciones:

```
   var sueldos;
  sueldos=new Array(5);
  cargar(sueldos);
  calcularGastos(sueldos);
```

Lo primero, definimos una variable y posteriormente creamos un objeto de la clase Array, indicándole que queremos almacenar 5 valores.
Llamamos a la función cargar enviándole el vector. En la función, a través de un ciclo for recorremos las distintas componentes del vector y almacenamos valores enteros que ingresamos por teclado.
Para conocer el tamaño del vector accedemos a la propiedad length de la clase Array.
En la segunda función sumamos todas las componentes del vector, imprimimos en la página los valores y el total de gastos.

Ejemplo 2: Crear un array con elementos de tipo string. Almacenar los meses de año. En otra función solicitar el ingreso de un número entre 1 y 12. Mostrar a qué mes corresponde y cuántos días tiene dicho mes.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function mostrarFecha(meses,dias)
  {
    var num;
    num=prompt('Ingrese número de mes:','');
    num=parseInt(num);
    document.write('Corresponde al mes:'+meses[num-1]);
    document.write('<br>');
    document.write('Tiene '+dias[num-1]+' días');
```

```
  }
  var meses;
  meses=new Array(12);
  meses[0]='Enero';
  meses[1]='Febrero';
  meses[2]='Marzo';
  meses[3]='Abril';
  meses[4]='Mayo';
  meses[5]='Junio';
  meses[6]='Julio';
  meses[7]='Agosto';
  meses[8]='Septiembre';
  meses[9]='Octubre';
  meses[10]='Noviembre';
  meses[11]='Diciembre';
  var dias;
  dias=new Array(12);
  dias[0]=31;
  dias[1]=28;
  dias[2]=31;
  dias[3]=30;
  dias[4]=31;
  dias[5]=30;
  dias[6]=31;
  dias[7]=31;
  dias[8]=30;
  dias[9]=31;
  dias[10]=30;
  dias[11]=31;
  mostrarFecha(meses,dias);
</script>
</body>
</html>
```

En este problema definimos dos vectores, uno para almacenar los meses y otro los días. Decimos que se trata de vectores paralelos porque en la componente cero del vector meses almacenamos el string 'Enero' y en el vector dias, la cantidad de días del mes de enero.
Es importante notar que cuando imprimimos, disponemos como subíndice el valor ingresado menos 1, esto debido a que normalmente el operador de nuestro programa carga un valor comprendido entre 1 y 12. Recordar que los vectores comienzan a numerarse a partir de la componente cero.

```
   document.write('Corresponde al mes:'+meses[num-1]);
```

7.5.3 BORRADO

Como los arrays de JavaScript son realmente objetos, podemos utilizar el operador delete para eliminar elementos de un array:

```
delete numbers[2];
// numbers es ['zero', 'one', undefined, 'shi', 'go']
```

Desafortunadamente, esto deja un espacio en el array. Esto es porque los elementos a la derecha del elemento eliminado conservan sus nombres. Para este caso, JavaScript incorpora una función splice, que permite eliminar y reemplazar elementos de un array. El primer argumento indica el por qué elemento comenzar a reemplazar, y el segundo argumento el número de elementos a eliminar.

```
numbers.splice(2,1);
// numbers es ['zero', 'one', 'shi', 'go']
```

7.5.4 OTRAS FUNCIONES UTILES PARA ARRAYS

A continuación se muestran las funciones más útiles para el manejo de arrays:

concat(), se emplea para concatenar los elementos de varios arrays

```
var array1 = [1, 2, 3];
array2 = array1.concat(4, 5, 6);   // array2 = [1, 2, 3, 4, 5, 6]
array3 = array1.concat([4, 5, 6]); // array3 = [1, 2, 3, 4, 5, 6]
```

join(separador), es la función contraria a split(). Une todos los elementos de un array para formar una cadena de texto. Para unir los elementos se utiliza el caracter separador indicado

```
var array = ["hola", "mundo"];
var mensaje = array.join(""); // mensaje = "holamundo"
mensaje = array.join(" ");    // mensaje = "hola mundo"
```

pop(), elimina el último elemento del array y lo devuelve. El array original se modifica y su longitud disminuye en 1 elemento.

```
var array = [1, 2, 3];
var ultimo = array.pop();
// ahora array = [1, 2], ultimo = 3
```

push(), añade un elemento al final del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
var array = [1, 2, 3];
array.push(4);
// ahora array = [1, 2, 3, 4]
```

shift(), elimina el primer elemento del array y lo devuelve. El array original se ve modificado y su longitud disminuida en 1 elemento.

```
var array = [1, 2, 3];
var primero = array.shift();
// ahora array = [2, 3], primero = 1
```

unshift(), añade un elemento al principio del array. El array original se modifica y aumenta su longitud en 1 elemento. (También es posible añadir más de un elemento a la vez)

```
var array = [1, 2, 3];
array.unshift(0);
// ahora array = [0, 1, 2, 3]
```

reverse(), modifica un array colocando sus elementos en el orden inverso a su posición original:

```
var array = [1, 2, 3];
array.reverse();
// ahora array = [3, 2, 1]
```

7.5.5 Arrays multidimensionales

Ahora veremos qué son los arrays multidimensionales (arrays de más de una dimensión) y cómo utilizarlos. Además explicaremos cómo inicializar arrays en su declaración.

Como estamos viendo, los arrays son bastante importantes en Javascript y también en la mayoría de los lenguajes de programación. En concreto ya hemos aprendido a crear arrays y utilizarlos.

Pero aún nos quedan algunas cosas importantes que explicar, como son los arrays de varias dimensiones.

Los arrays multidimensionales son estructuras de datos que almacenan los valores en más de una dimensión. Los arrays que hemos visto hasta ahora almacenan valores en una dimensión, por eso para acceder a las posiciones utilizamos tan solo un índice. Los arrays de 2 dimensiones guardan sus valores, por decirlo de alguna manera, en filas y columnas y por ello necesitaremos dos índices para

acceder a cada una de sus posiciones.

Dicho de otro modo, un array multidimensional es como un contenedor que guardara más valores para cada posición, es decir, como si los elementos del array fueran a su vez otros arrays.

En Javascript no existe un auténtico objeto array-multidimensinal. Para utilizar estas estructuras podremos definir arrays que donde en cada una de sus posiciones habrá otro array. En nuestros programas podremos utilizar arrays de cualquier dimensión, veremos a continuación cómo trabajar con arrays de dos dimensiones, que serán los más comunes.

En este ejemplo vamos a crear un array de dos dimensiones donde tendremos por un lado ciudades y por el otro la temperatura media que hace en cada una durante de los meses de invierno.

```
var temperaturas_medias_ciudad0 = new Array(3)
temperaturas_medias_ciudad0[0] = 12
temperaturas_medias_ciudad0[1] = 10
temperaturas_medias_ciudad0[2] = 11

var temperaturas_medias_ciudad1 = new Array (3)
temperaturas_medias_ciudad1[0] = 5
temperaturas_medias_ciudad1[1] = 0
temperaturas_medias_ciudad1[2] = 2

var temperaturas_medias_ciudad2 = new Array (3)
temperaturas_medias_ciudad2[0] = 10
temperaturas_medias_ciudad2[1] = 8
temperaturas_medias_ciudad2[2] = 10
```

Con las anteriores líneas hemos creado tres arrays de 1 dimensión y tres elementos, como los que ya conocíamos. Ahora crearemos un nuevo array de tres elementos e introduciremos dentro de cada una de sus casillas los arrays creados anteriormente, con lo que tendremos un array de arrays, es decir, un array de 2 dimensiones.

```
var temperaturas_cuidades = new Array (3)
temperaturas_cuidades[0] = temperaturas_medias_ciudad0
temperaturas_cuidades[1] = temperaturas_medias_ciudad1
temperaturas_cuidades[2] = temperaturas_medias_ciudad2
```

Vemos que para introducir el array entero hacemos referencia al mismo sin paréntesis ni corchetes, sino sólo con su nombre. El array temperaturas_cuidades es nuestro array bidimensional.

También es interesante ver cómo se realiza un recorrido por un array de dos dimensiones. Para ello tenemos que hacer un bucle que pase por cada una de las casillas del array bidimensional y dentro de éstas hacer un nuevo recorrido para cada una de sus casillas internas. Es decir, un recorrido por un array dentro de otro.

El método para hacer un recorrido dentro de otro es colocar un bucle dentro de otro, lo que se llama un bucle anidado. En este ejemplo vamos a meter un bucle FOR dentro de otro. Además, vamos a escribir los resultados en una tabla, lo que complicará un poco el script, pero así podremos ver cómo construir una tabla desde Javascript a medida que realizamos el recorrido anidado al bucle.

```
document.write("<table width=200 border=1 cellpadding=1 cellspacing=1>");
for (i=0;i<temperaturas_cuidades.length;i++){
    document.write("<tr>")
    document.write("<td><b>Ciudad " + i + "</b></td>")
    for (j=0;j<temperaturas_cuidades[i].length;j++){
       document.write("<td>" + temperaturas_cuidades[i][j] + "</td>")
    }
    document.write("</tr>")
}
document.write("</table>")
```

Este script resulta un poco más complejo que los vistos anteriormente. La primera acción consiste en escribir la cabecera de la tabla, es decir, la etiqueta <TABLE> junto con sus atributos. Con el primer bucle realizamos un recorrido a la primera dimensión del array y utilizamos la variable i para llevar la

cuenta de la posición actual. Por cada iteración de este bucle escribimos una fila y para empezar la fila abrimos la etiqueta <TR>. Además, escribimos en una casilla el numero de la ciudad que estamos recorriendo en ese momento. Posteriormente ponemos otro bucle que va recorriendo cada una de las casillas del array en su segunda dimensión y escribimos la temperatura de la ciudad actual en cada uno de los meses, dentro de su etiqueta <TD>. Una vez que acaba el segundo bucle se han impreso las tres temperaturas y por lo tanto la fila está terminada. El primer bucle continúa repitiéndose hasta que todas las ciudades están impresas y una vez terminado cerramos la tabla.

Inicialización de Arrays

Para terminar con el tema de los arrays vamos a ver una manera de inicializar sus valores a la vez que lo declaramos, así podemos realizar de una manera más rápida el proceso de introducir valores en cada una de las posiciones del array.

El método normal de crear un array vimos que era a través del objeto Array, poniendo entre paréntesis el número de casillas del array o no poniendo nada, de modo que el array se crea sin ninguna posición. Para introducir valores a un array se hace igual, pero poniendo entre los paréntesis los valores con los que deseamos rellenar las casillas separados por coma. Veámoslo con un ejemplo que crea un array con los nombres de los días de la semana.

```
var diasSemana = new
Array("Lunes","Martes","Miércoles,","Jueves","Viernes","Sábado","Domingo")
```

El array se crea con 7 casillas, de la 0 a la 6 y en cada casilla se escribe el día de la semana correspondiente (Entre comillas porque es un texto).

Ahora vamos a ver algo más complicado, se trata de declarar el array bidimensional que utilizamos antes para las temperaturas de las ciudades en los meses en una sola línea, introduciendo los valores a la vez.

```
var temperaturas_cuidades = new Array(new Array (12,10,11), new Array(5,0,2),new
Array(10,8,10))
```

En el ejemplo introducimos en cada casilla del array otro array que tiene como valores las temperaturas de una ciudad en cada mes.

Javascript todavía tiene una manera más resumida que la que acabamos de ver, que explicamos cuando tratamos los arrays unidimensionales. Para ello simplemente escribimos entre corchetes los datos del array que estamos creando. Para acabar vamos a mostrar un ejemplo sobre cómo utilizar esta sintaxis para declarar arrays de más de una dimensión.

```
var arrayMuchasDimensiones = [1, ["hola", "que", "tal", ["estas", "estamos",
"estoy"], ["bien", "mal"], "acabo"], 2, 5];
```

En este ejemplo hemos creado un array muy poco uniforme, porque tiene casillas con contenido de simples enteros y otras con contenido de cadena y otras que son otros arrays. Podríamos acceder a algunas de sus casillas y mostrar sus valores de esta manera:

```
alert (arrayMuchasDimensiones[0])
alert (arrayMuchasDimensiones[1][2])
alert (arrayMuchasDimensiones[1][3][1])
```

```
Ejemplos de Arrray y funciones de fecha
<SCRIPT>
var diasemana=new Array(7);
diasemana[0]="Domingo";
diasemana[1]="Lunes";
diasemana[2]="Martes";
diasemana[3]="Miércoles";
diasemana[4]="Jueves";
diasemana[5]="Viernes";
diasemana[6]="Sábado";
var fecha=new Date();
var ldiasemana=diasemana[fecha.getDay()];
document.write(ldiasemana);
</SCRIPT>

<SCRIPT>
var months=new Array(13);
months[1]="Enero";
months[2]="Febrero";
months[3]="Marzo";
months[4]="Abril";
```

```
months[5]="Mayo";
months[6]="Junio";
months[7]="Julio";
months[8]="Agosto";
months[9]="Septiembre";
months[10]="Octubre";
months[11]="Noviembre";
months[12]="Diciembre";
var time=new Date();
var lmonth=months[time.getMonth() + 1];
var date=time.getDate();
var year=time.getYear();
if (year < 2000)
    year = year + 1900;
    document.write(+ date + " de ");
    document.write(lmonth + " de " + year);
</SCRIPT>
```

CAPÍTULO 8: FORMULARIOS Y EVENTOS

El uso de JavaScript en los formularios HTML se hace fundamentalmente con el objetivo de validar los datos ingresados. Se hace esta actividad en el cliente (navegador) para desligar de esta actividad al servidor que recibirá los datos ingresados por el usuario.
Esta posibilidad de hacer pequeños programas que se ejecutan en el navegador, evitan intercambios innecesarios entre el cliente y el servidor (navegador y sitio web).
En esta instancia damos por conocidas las marcas para la creación de formularios en una página web:

```
  form <form> ... </form>
  text <input type="text">
  password <input type="password">
  textarea <textarea> ... </textarea>
  button <input type="button">
  submit <input type="submit">
  reset <input type="reset">
  checkbox <input type="checkbox">
  radio <input type="radio">
  select <select> ... </select>
  hidden <input type="hidden">
```

El navegador crea un objeto por cada control visual que aparece dentro de la página. Nosotros podemos acceder posteriormente desde JavaScript a dichos objetos.
El objeto principal es el FORM que contendrá todos los otros objetos: TEXT (editor de líneas), TEXTAREA (editor de varias líneas), etc.
Nuestra actividad en JavaScript es procesar los eventos que generan estos controles (un evento es una acción que se dispara, por ejemplo si se presiona un botón).
Vamos a hacer un problema muy sencillo empleando el lenguaje JavaScript; dispondremos un botón y cada vez que se presione **-evento onClick()**-, mostraremos un contador:

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
var contador=0;
function incrementar(){
  contador++;
  alert('El contador ahora vale :' + contador);
```

```
}
</script>
<form>
  <input type="button" onClick="incrementar()" value="incrementar">
</form>
</body>
</html>
```

A los eventos de los objetos HTML se les asocia una función, dicha función se ejecuta cuando se dispara el evento respectivo. En este caso cada vez que presionamos el botón, se llama a la función incrementar, en la misma incrementamos la variable contador en uno. Hay que tener en cuenta que a la variable contador la definimos fuera de la función para que no se inicialice cada vez que se dispara el evento.
La función alert crea una ventana que puede mostrar un mensaje.

Nota: el ejemplo anterior pudo haberse resuelto sin funciones, de la forma:

```
<html>
<head>
</head>
<body>
<script>
 var contador=0;
</script>
<form>
  <input type="button" value="incrementar"
onclick="contador++;alert('El contador ahora vale ' + contador);">
</form>
</body>
</html>
```

Esto es, incluir el código JavaScript dentro de atributos HTML.

Se trata del método más sencillo y a la vez menos profesional de indicar el código JavaScript que se debe ejecutar cuando se produzca un evento. Sólo utilizable para acciones puntuales y muy particulares, sino, recurrir a funciones, y en lo posible, que dichas funciones residan en un archivo externo .js.

8.1 Controles FORM, BUTTON y TEXT – Evento onClick

Hasta ahora hemos visto como crear un formulario con controles de tipo BUTTON. Agregamos un control de tipo TEXT (permite al operador cargar caracteres por teclado).
Ahora veremos la importancia de definir un id a todo control de un formulario.
Con un ejemplo veremos estos controles: Confeccionar un formulario que permita ingresar el nombre y edad de una persona:

```
<html>
<head></head>
<body>
<script type="text/javascript">
function mostrar(){
  var nom=document.getElementById('nombre').value;
  var ed=document.getElementById('edad').value;
  alert('Ingresó el nombre:' + nom);
  alert('Y la edad:' + ed);
}
</script>
<form>
Ingrese su nombre: <input type="text" id="nombre"><br>
Ingrese su edad: <input type="text" id="edad"><br>
<input type="button" value="Confirmar" onClick="mostrar()">
</form>
</body>
</html>
```

En este problema tenemos tres controles: 1-FORM, 2-BUTTON, 3-TEXT. El evento que se dispara al presionar el botón se llama onClick.
La función 'mostrar()' accede a los contenidos de los dos controles de tipo TEXT:

```
  var nom=document.getElementById('nombre').value;
  var ed=document.getElementById('edad').value;
```

Para hacer más clara la función guardamos en dos variables auxiliares los contenidos de los controles de tipo TEXT.
La propiedad "id" es un identificar único para cualquier marca HTML que luego nos permite desde Javascript acceder a dicho elemento.
El método **getElementById** nos retorna una referencia del objeto HTML que le pasamos como

parámetro. A partir de este objeto accedemos a la propiedad value que almacena el valor ingresado por el operador en el control TEXT.
Luego de extraer los valores ingresados por el operador los mostramos utilizando la función alert:

```
  var nom=document.getElementById('nombre').value;
  var ed=document.getElementById('edad').value;
  alert('Ingresó el nombre:' + nom);
  alert('Y la edad:' + ed);
```

8.2 Control PASSWORD

Esta marca es una variante de la de tipo "TEXT". La diferencia fundamental es que cuando se carga un texto en el campo de edición sólo muestra asteriscos en pantalla, es decir, es fundamental para el ingreso de claves y para que otros usuarios no vean los caracteres que tipeamos.

La mayoría de las veces este dato se procesa en el servidor. Pero podemos en el cliente (es decir en el navegador) verificar si ha ingresado una cantidad correcta de caracteres, por ejemplo.

Ejemplo: Codificar una página que permita ingresar una password y luego muestre una ventana de alerta si tiene menos de 5 caracteres.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function verificar() {
    var clave=document.getElementById('clave').value;
    if (clave.length<5) {
      alert('La clave no puede tener menos de 5 caracteres!!!');
    }
    else {
      alert('Largo de clave correcta');
    }
  }
</script>
<form>
Ingrese una clave:
```

```
<input type="password" id="clave">
<br>
<input type="button" value="Confirmar" onClick="verificar()">
</form>
</body>
</html>
```

En este problema debemos observar que cuando ingresamos caracteres dentro del campo de edición sólo vemos asteriscos, pero realmente en memoria se almacenan los caracteres tipeados. Si queremos mostrar los caracteres ingresados debemos acceder mediante el método getElementById a la marca HTML clave:

```
    var clave=document.getElementById('clave').value;
```

Normalmente, a este valor no lo mostraremos dentro de la página, sino se perdería el objetivo de este control (ocultar los caracteres tipeados).
Si necesitamos saber la cantidad de caracteres que tiene un string accedemos a la propiedad length que retorna la cantidad de caracteres.

```
    if (clave.length<5)
```

8.3 Control TEXTAREA

Este control es similar al control TEXT, salvo que permite el ingreso de muchas líneas de texto.
La marca TEXTAREA en HTML tiene dos propiedades: rows y cols que nos permiten indicar la cantidad de filas y columnas a mostrar en pantalla.
Ejemplo: Solicitar la carga del mail y el curriculum de una persona. Mostrar un mensaje si el curriculum supera los 2000 caracteres.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
function controlarCaracteres(){
  if (document.getElementById('curriculum').value.length>2000) {
    alert('curriculum muy largo');
  }
  else {
    alert('datos correctos');
```

```
  }
}
</script>
<form>
<textarea id="curriculum" rows="10" cols="50" ></textarea>
<br>
<input type="button" value="Mostrar" onClick="controlarCaracteres()">
</form>
</body>
</html>
```

Para saber el largo de la cadena cargada accedemos a la propiedad length:

```
  If (document.getElementById('curriculum').value.length>2000)
```

8.4 Control CHECKBOX

El control CHECKBOX es el cuadradito que puede tener dos estados (seleccionado o no seleccionado).
Para conocer su funcionamiento y ver cómo podemos acceder a su estado desde Javascript haremos una pequeña página.
Ejemplo: Confeccionar una página que muestre 4 lenguajes de programación que el usuario puede seleccionar si los conoce. Luego mostrar un mensaje indicando la cantidad de lenguajes que ha seleccionado el operador.

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
function contarSeleccionados()
{
  var cant=0;
  if (document.getElementById('checkbox1').checked){
    cant++;
  }
  if (document.getElementById('checkbox2').checked){
    cant++;
```

```
  }
  if (document.getElementById('checkbox3').checked){
    cant++;
  }
  if (document.getElementById('checkbox4').checked){
    cant++;
  }
  alert('Conoce ' + cant + ' lenguajes');
}
</script>

<form>
<input type="checkbox" id="checkbox1">JavaScript
<br>
<input type="checkbox" id="checkbox2">PHP
<br>
<input type="checkbox" id="checkbox3">JSP
<br>
<input type="checkbox" id="checkbox4">VB.Net
<br>
<input type="button" value="Mostrar" onClick="contarSeleccionados()">
</form>
</body>
</html>
```

Cuando se presiona el botón se llama a la función Javascript contarSeleccionados(). En la misma verificamos uno a uno cada control checkbox accediendo a la propiedad checked del elemento que almacena true o false según esté o no seleccionado el control:
Disponemos un 'if' para cada checkbox:

```
  if (document.getElementById('checkbox1').checked){
    cant++;
  }
```

Como la propiedad checked almacena un true o false podemos utilizar dicho valor directamente como valor de la condición en lugar de codificar:

```
  if (document.getElementById('checkbox1').checked==true){
```

```
    cant++;
  }
```

Al contador 'cant' lo definimos e inicializamos en cero previo a los cuatro if. Mostramos finalmente el resultado final.

8.5 Control RADIO

Los objetos RADIO tienen sentido cuando disponemos varios elementos. Sólo uno puede estar seleccionado del conjunto.
Ejemplo: Mostrar cuatro objetos de tipo RADIO que permitan seleccionar los estudios que tiene un usuario:

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
  function mostrarSeleccionado() {
    if (document.getElementById('radio1').checked) {
      alert('no tienes estudios');
    }
    if (document.getElementById('radio2').checked) {
      alert('tienes estudios primarios');
    }
    if (document.getElementById('radio3').checked) {
      alert('tienes estudios secundarios');
    }
    if (document.getElementById('radio4').checked) {
      alert('tienes estudios universitarios');
    }
  }
</script>
<form>
<input type="radio" id="radio1" name="estudios">Sin estudios
<br>
<input type="radio" id="radio2" name="estudios">Primarios
```

```
<br>
<input type="radio" id="radio3" name="estudios">Secundarios
<br>
<input type="radio" id="radio4" name="estudios">Universitarios
<br>
<input type="button" value="Mostrar" onClick="mostrarSeleccionado()">
</form>
</body>
</html>
```

Es importante notar que todos los objetos de tipo RADIO tienen definida la propiedad name con el mismo valor (esto permite especificar que queremos que los radios estén relacionados entre si) Luego podemos acceder a cada elemento mediante el método getElementById para consultar la propiedad checked:

```
    if (document.getElementById('radio1').checked) {
       alert('no tienes estudios');
     }
```

Igual que el checkbox, la propiedad checked retorna true o false, según esté o no seleccionado el control radio.

8.6 Control SELECT - Evento onChange

Este otro objeto visual que podemos disponer en un FORM permite realizar la selección de un string de una lista y tener asociado al mismo un valor no visible. El objetivo fundamental en JavaScript es determinar qué elemento está seleccionado y qué valor tiene asociado. Esto lo hacemos cuando ocurre el evento *onChange*.

Para determinar la posición del índice seleccionado en la lista:

```
document.getElementById('select1').selectedIndex;
```

Considerando que el objeto SELECT se llama select1 accedemos a la propiedad selectedIndex (almacena la posición del string seleccionado de la lista, numerando a partir de cero).

Para determinar el string seleccionado:

```
document.getElementById('select1').options[document.getElementById('se
lect1').selectedIndex].text;
```

Es decir que el objeto select1 tiene otra propiedad llamada options, a la que accedemos por medio de un subíndice, al string de una determinada posición.

Hay problemas en los que solamente necesitaremos el string almacenado en el objeto SELECT y no

el valor asociado (no es obligatorio asociar un valor a cada string).

Y por último con esta expresión accedemos al valor asociado al string:

```
document.getElementById('select1').options[document.getElementById('se
lect1').selectedIndex].value;
```

Un ejemplo completo que muestra el empleo de un control SELECT es:

```
<html>
<head>
</head>
<body>
<script type="text/javascript">
function cambiarColor(){
  var seleccion=document.getElementById('select1');
  document.getElementById('text1').value=seleccion.selectedIndex;
document.getElementById('text2').value=seleccion.options[seleccion.sel
ectedIndex].text;
document.getElementById('text3').value=seleccion.options[seleccion.sel
ectedIndex].value;
}
</script>
<form>
  <select id="select1" onChange="cambiarColor()">
    <option value="0xff0000">Rojo</option>
    <option value="0x00ff00">Verde</option>
    <option value="0x0000ff">Azul</option>
  </select>
  <br>
  Número de índice seleccionado del objeto SELECT:<input type="text"
id="text1"><br>
  Texto seleccionado:<input type="text" id="text2"><br>
  Valor asociado:<input type="text" id="text3"><br>
</form>
</body>
</html>
```

Se debe analizar en profundidad este problema para comprender primeramente la creación del objeto SELECT en HTML, y cómo acceder luego a sus valores desde Javascript.
Analizando la función cambiarColor() podemos ver cómo obtenemos los valores fundamentales del objeto SELECT.

8.7 Otros Eventos

8.7.1 Eventos onFocus y onBlur

El evento **onFocus** se dispara cuando el objeto toma foco y el evento **onBlur** cuando el objeto pierde el foco.
Ejemplo: Implementar un formulario que solicite la carga del nombre y la edad de una persona. Cuando el control tome foco borrar el contenido actual, al abandonar el mismo, mostrar un mensaje de alerta si el mismo está vacío. Mostrar en las propiedades value de los controles text los mensajes "nombre" y "mail" respectivamente.

```
<html>
<head></head>
<body>
<script type="text/javascript">
function vaciar(control) {
  control.value='';
}
function verificarEntrada(control) {
  if (control.value=='') alert('Debe ingresar datos');
}
</script>
<form>
<input type="text" id="nombre" onFocus="vaciar(this)"
onBlur="verificarEntrada(this)" value="nombre"><br>
<input type="text" id="edad" onFocus="vaciar(this)"
onBlur="verificarEntrada(this)" value="mail">
<br>
<input type="button" value="Confirmar">
```

```
</form>
</body>
</html>
```

A cada control de tipo TEXT le inicializamos los eventos **onFocus** y **onBlur**. También cargamos las propiedad value para mostrar un texto dentro del control. Le indicamos, para el evento **onFocus** la función vaciar, pasando como parámetro la palabra clave **this** que significa la dirección del objeto que emitió el evento. En la función propiamente dicha, accedemos a la propiedad **value** y borramos su contenido. Esto nos permite definir una única función para vaciar los dos controles. De forma similar, para el evento **onBlur** llamamos a la función **verificarEntrada** donde analizamos si se ha ingresado algún valor dentro del control, en caso de tener un string vacío procedemos a mostrar una ventana de alerta.

8.7.2 Eventos onMouseOver y onMouseOut

El evento **onMouseOver** se ejecuta cuando pasamos la flecha del mouse sobre un elemento HTML y el evento **onMouseOut** cuando la flecha abandona el mismo.
Para probar estos eventos implementaremos una página que cambie el color de fondo del documento. Implementaremos una función que cambie el color con un valor que llegue como parámetro. Cuando retiramos la flecha del mouse volvemos a pintar de blanco el fondo del documento:

```
<html>
<head></head>
<body>
<script type="text/javascript">
function pintar(col) {
    document.bgColor=col;
}
</script>
<a href="pagina1.html" onMouseOver="pintar('#ff0000')"
onMouseOut="pintar('#ffffff')">Rojo</a>
-
<a href="pagina1.html" onMouseOver="pintar('#00ff00')"
onMouseOut="pintar('#ffffff')">Verde</a>
-
<a href="pagina1.html" onMouseOver="pintar('#0000ff')"
onMouseOut="pintar('#ffffff')">Azul</a>
```

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

Las llamadas a las funciones las hacemos inicializando las propiedades onMouseOver y onMouseOut:

```
<a href="pagina1.html" onMouseOver="pintar('#ff0000')"
onMouseOut="pintar('#ffffff')">Rojo</a>
```

La función 'pintar' recibe el color e inicializa la propiedad bgColor del objeto document.
function pintar(col)

```
{
  document.bgColor=col;
}
```

Otro problema que podemos probar es pinta de color el interior de una casilla de una tabla y regresar a su color original cuando salimos de la misma:

```
<html>
<head></head>
<body>
<script type="text/javascript">
function pintar(objeto,col) {
    objeto.bgColor=col;
}
</script>
<table border="1">
<tr>
<td onMouseOver="pintar(this,'#ff0000')"
onMouseOut="pintar(this,'#ffffff')">rojo</td>
<td onMouseOver="pintar(this,'#00ff00')"
onMouseOut="pintar(this,'#ffffff')">verde</td>
<td onMouseOver="pintar(this,'#0000ff')"
onMouseOut="pintar(this,'#ffffff')">azul</td>
</tr>
</table>
</body>
</html>
```

La lógica es bastante parecida a la del primer problema, pero en éste, le pasamos como parámetro a la función, la referencia a la casilla que queremos que se coloree (this):

```
<td onMouseOver="pintar(this,'#ff0000')"
onMouseOut="pintar(this,'#ffffff')">rojo</td>
```

8.7.3 Evento onLoad

El evento **onLoad** se ejecuta cuando cargamos una página en el navegador. Uno de los usos más frecuentes es para fijar el foco en algún control de un formulario, para que el operador no tenga que activar con el mouse dicho control.
Este evento está asociado a la marca body.
La página completa es:

```
<html>
<head></head>
<body onLoad="activarPrimerControl()">
<script type="text/javascript">
function activarPrimerControl() {
    document.getElementById('nombre').focus();
}
</script>
<form>
  Ingrese su nombre:
  <input type="text" id="nombre"><br>
  Ingrese su edad:
  <input type="text" id="edad"><br>
  <input type="button" value="Confirmar">
</form>
</body>
</html>
```

En la marca body inicializamos el evento onLoad con la llamada a la función activarPrimerControl:

```
<body onLoad="activarPrimerControl()">
```

La función da el foco al control text donde se cargará el nombre:

```
 function activarPrimerControl(){
    document.getElementById('nombre').focus();
```

```
}
```

La siguiente tabla resume los eventos más importantes definidos por JavaScript:

Evento	Descripción	Elementos para los que está definido
`Onblur`	Deseleccionar el elemento	`<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`, `<body>`
`Onchange`	Deseleccionar un elemento que se ha modificado	`<input>`, `<select>`, `<textarea>`
`Onclick`	Pinchar y soltar el ratón	Todos los elementos
`Ondblclick`	Pinchar dos veces seguidas con el ratón	Todos los elementos
`Onfocus`	Seleccionar un elemento	`<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`, `<body>`
`Onkeydown`	Pulsar una tecla (sin soltar)	Elementos de formulario y `<body>`
`Onkeypress`	Pulsar una tecla	Elementos de formulario y `<body>`
`Onkeyup`	Soltar una tecla pulsada	Elementos de formulario y `<body>`

Evento	Descripción	Elementos para los que está definido
`Onload`	La página se ha cargado completamente	`<body>`
`Onmousedown`	Pulsar (sin soltar) un botón del ratón	Todos los elementos
`Onmousemove`	Mover el ratón	Todos los elementos
`Onmouseout`	El ratón *"sale"* del elemento (pasa por encima de otro elemento)	Todos los elementos
`Onmouseover`	El ratón *"entra"* en el elemento (pasa por encima del elemento)	Todos los elementos
`Onmouseup`	Soltar el botón que estaba pulsado en el ratón	Todos los elementos
`Onreset`	Inicializar el formulario (borrar todos sus datos)	`<form>`
`Onresize`	Se ha modificado el tamaño de la ventana del navegador	`<body>`
`Onselect`	Seleccionar un texto	`<input>`, `<textarea>`

Evento	Descripción	Elementos para los que está definido
`Onsubmit`	Enviar el formulario	`<form>`
`Onunload`	Se abandona la página (por ejemplo al cerrar el navegador)	`<body>`

Nota: Manejadores de eventos y variable *this*

JavaScript define una variable especial llamada this que se crea automáticamente y que se emplea en algunas técnicas avanzadas de programación. En los eventos, se puede utilizar la variable this para referirse al elemento HTML que ha provocado el evento. Esta variable es muy útil para ejemplos como el siguiente:

Cuando el usuario pasa el ratón por encima del , el color del borde se muestra de color negro. Cuando el ratón *sale* del , se vuelve a mostrar el borde con el color gris claro original.

Elemento original:

```
<div id="contenidos" style="width:150px; height:60px; border:thin
solid silver">
  Sección de contenidos...
</div>
```

Si no se utiliza la variable ***this***, el código necesario para modificar el color de los bordes, sería el siguiente:

```
<div id="contenidos" style="width:150px; height:60px; border:thin
solid silver"
onmouseover="document.getElementById('contenidos').style.borderColor='
black';"
onmouseout="document.getElementById('contenidos').style.borderColor='s
ilver';">
  Sección de contenidos...
</div>
```

El código anterior es demasiado largo y demasiado propenso a cometer errores. Dentro del código de un evento, JavaScript crea automáticamente la variable this, que hace referencia al elemento HTML que ha provocado el evento. Así, el ejemplo anterior se puede reescribir de la siguiente manera:

```
<div id="contenidos" style="width:150px; height:60px; border:thin
solid silver" onmouseover="this.style.borderColor='black';"
onmouseout="this.style.borderColor='silver';">
  Sección de contenidos...
</div>
```

El código anterior es mucho más compacto, más fácil de leer y de escribir y sigue funcionando correctamente aunque se modifique el valor del atributo id del .

El principal inconveniente de este método es que en las funciones externas no se puede seguir utilizando la variable ***this*** y por tanto, es necesario pasar esta variable como parámetro a la función.

8.8 Envío (submit) del formulario

En los formularios existe habitualmente un botón que permite el envío del contenido del mismo al destino fijado en el atributo ACTION de la directiva <FORM>. Este botón se define como <INPUT TYPE="SUBMIT" VALUE="Enviar">. La activación de este botón produce el envío del formulario, pero con anterioridad al mismo se produce un evento **onSubmit**, que puede ser tratado con un manejador de evento colocado en la directiva <FORM>.

Las expresiones JavaScript colocadas en el manejador del evento **onSubmit** son ejecutadas antes de proceder al envío, si el resultado de dichas expresiones, como el valor retornado por una función, es TRUE, el envío se produce sin más, pero si ese valor retornado es FALSE, el envío se cancela.

Esto permite que se pueda realizar un chequeo y comprobación de los datos antes del envío, cancelando el mismo si los datos no son correctos, sin perder las validaciones automáticas del submit en HTML5.

```
<html>
<head>

<script>
  function valida() {
        alert("nuevas validaciones");
        return true; //o false
  }
</script>

</head>
<body>

<form onSubmit="return valida()" method="POST" action="procesar.php">
   nombre:<input type="text" size="30" autofocus required>
   <br></br>
   <input type="submit" value="Enviar Nombre">
</form>

</body>
</html>
```

Programación orientada a objetos en JavaScrip

Retomando el tema de la programación orientada a objetos, recordemos que el lenguaje JavaScript no es un lenguaje orientado a objetos completo, pero permite definir clases con sus atributos y responsabilidades. Finalmente nos permite definir objetos de estas clases.
Pero el otro pilar de la programación orientada a objetos, es decir la herencia, no está implementada en el lenguaje.
Veremos la sintaxis para la declaración de una clase y la posterior definición de objetos de la misma.

Desarrollaremos una clase que represente un cliente de un banco.
La clase cliente tiene como atributos:

```
nombre
saldo
```

y las responsabilidades o métodos de la clase son:

```
Constructor (inicializamos los atributos del objeto)
depositar
extraer
```

Luego debemos implementar los siguientes métodos (normalmente el constructor se utiliza el caracter mayúscula):

```
function Cliente(nombre,saldo){
  this.nombre=nombre;
  this.saldo=saldo;
  this.depositar=depositar;
  this.extraer=extraer;
}

function depositar(dinero){
  this.saldo=this.saldo+dinero;
}

function extraer(dinero){
  this.saldo=this.saldo-dinero;
}
```

El nombre de la clase coincide con el nombre de la función principal que implementamos (también llamado constructor de la clase):

```
function cliente(nombre,saldo){
  this.nombre=nombre;
  this.saldo=saldo;
```

```
  this.depositar=depositar;
  this.extraer=extraer;
}
```

A esta función llegan como parámetro los valores con que queremos inicializar los atributos. Con la palabra clave 'this' diferenciamos los atributos de los parámetros (los atributos deben llevar la palabra clave this)

```
  this.nombre=nombre;
  this.saldo=saldo;
```

También en el constructor inicializamos la referencia a todos los métodos que contendrá la clase (esto es muy importante y necesario para entender porque las otras dos funciones pertenecen a esta clase):

```
  this.depositar=depositar;
  this.extraer=extraer;
```

Por último, implementamos todos los métodos de la clase:

```
function depositar(dinero){
  this.saldo=this.saldo+dinero;
}

function extraer(dinero){
  this.saldo=this.saldo-dinero;
}
```

De nuevo recordemos que diferenciamos los atributos de la clase por la palabra clave **this**.

Ahora veamos el archivo HTML completo donde además definiremos un objeto de la clase planteada:

```
<html>
<head>
<title>Problema</title>

<script>
  function Cliente(nombre,saldo) {  //constructor de la clase
    this.nombre=nombre;
    this.saldo=saldo;
    this.depositar=depositar;
    this.extraer=extraer;
  }

  function depositar(dinero) {    //método depositar
    this.saldo=this.saldo+dinero;
  }
```

```
  function extraer(dinero) {    //método extraer
    this.saldo=this.saldo-dinero;
  }

</script>

</head>

<body>
<script>
  var cliente1;
  cliente1 = new Cliente('diego',1200);
  document.write('Nombre del cliente:'+cliente1.nombre+'<br>');
  document.write('Saldo actual:'+cliente1.saldo+'<br>');
  cliente1.depositar(120);
  document.write('Saldo luego de depositar $120:>' +
  cliente1.saldo + '<br>');
  cliente1.extraer(1000);
  document.write('Saldo luego de extraer $1000:>' +
  cliente1.saldo+'<br>');
</script>

</body>
</html>
```

Hemos dividido la declaración de la clase en un bloque Javascript distinto a donde definimos un objeto de la misma, esto no es obligatorio, pero podemos ver que queda más claro.

Para definir un objeto de la clase Cliente tenemos:

```
  var cliente1;
  cliente1 = new Cliente('diego',1200);
```

Luego las llamadas a métodos le antecedemos el nombre del objeto llamado cliente1:

```
  document.write('Nombre del cliente:' +
  cliente1.nombre+'<br>');
  document.write('Saldo actual:' + cliente1.saldo + '<br>');
  cliente1.depositar(120);
  document.write('Saldo luego de depositar $120: ' +
  cliente1.saldo + '<br>');
  cliente1.extraer(1000);
```

```
  document.write('Saldo luego de extraer $1000: ' +
  cliente1.saldo + '<br>');
```

Podemos decir que la ventaja que podemos obtener con el planteo de clases es hacer nuestros programas mucho más organizados, entendibles y fundamentalmente, poder reutilizar clases en distintos proyectos.

Ejercicio: Confeccionar una clase llamada suma, que contenga dos atributos (valor1, valor2) y tres métodos: cargarValor1, cargarValor2 y retornarResultado. Implementar la clase suma.
La definición de un objeto de la clase que deben plantear es:

```
var s = new Suma();
s.primervalor(10);
s.segundovalor(20);
document.write('La suma de los dos valores es:' +
s.retornarresultado());
```

```
<html>
<head>

<script>
  function Suma(valor1,valor2) {
    this.valor1=valor1;
    this.valor2=valor2;
    this.primerValor=primerValor;
    this.segundoValor=segundoValor;
    this.retornarResultado=retornarResultado;
  }

  function primerValor(valor1) {
    this.valor1=valor1;
  }

  function segundoValor(valor2) {
    this.valor2=valor2;
  }

  function retornarResultado() {
```

```
    return this.valor1 + this.valor2;
  }
</script>

</head>
<body>

<script>
  var suma1;
  suma1 = new Suma(5,10);
  document.write('La suma de 5 y 10 es: ' +
  suma1.retornarResultado() + '<br>');
  suma1.primerValor(70);
  suma1.segundoValor(30);
  document.write('La suma de 70 y 30 es: ' +
  suma1.retornarResultado() + '<br>');
</script>

</body>
</html>
```