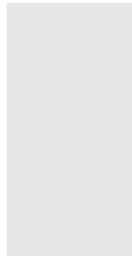




estudios abiertos

SEAS

GRUPO SANVALERO



Programación Web

1. Lenguaje JavaScript

ÍNDICE

OBJETIVOS	11
INTRODUCCIÓN	12
1.1. ¿Qué es JavaScript?	13
1.1.1. Historia	13
1.2. ¿Cómo incluir JavaScript en documentos XHTML?	15
1.2.1. Introducirlo en el mismo documento XHTML	15
1.2.2. Definir JavaScript en un archivo externo	15
1.2.3. Insertarlo en los elementos XHTML	17
1.3. La etiqueta noscript	18
1.4. Sintaxis	19
1.5. Posibilidades y limitaciones de JavaScript	21
1.5.1. JavaScript en los navegadores	21
1.6. Nuestro primer script	22
1.7. Elementos básicos	25
1.7.1. Variables y tipos de variables	25
1.7.2. Operadores	31
1.8. Estructuras de control de flujo	40
1.8.1. Estructura If, If...Else	40
1.8.2. Estructura switch	43
1.8.3. Estructura For	46
1.8.4. Estructura While	48
1.8.5. Estructura Do...While	50
1.8.6. Estructura For...In	52
1.8.7. Utilización de Arrays mediante estructuras de control	53
1.9. Funciones	61
1.9.1. Creación de una función	61
1.9.2. ¿Dónde declarar una función?	62
1.9.3. Llamada a una función	63
1.9.4. Argumentos	64
1.9.5. Consideraciones con el uso de funciones	65
1.9.6. Variables locales y globales	66
1.9.7. Funciones predefinidas	67
1.9.8. Funciones de cadena de texto	69
1.10. Objetos	71
1.10.1. Jerarquía de los objetos	72
1.10.2. Propiedades y métodos de los objetos del navegador	81
1.10.3. Propiedades y métodos de los objetos del documento	110
1.10.4. Propiedades y métodos de los objetos del lenguaje	134
RESUMEN	153

OBJETIVOS

- Conoceremos que es JavaScript y la forma en que surgió.
- Veremos como incluirlo en los documentos XHTML.
- Descubriremos su sintaxis y sus posibles limitaciones.
- Empezaremos por el principio:
 - Presentaremos todos los elementos básicos de su programación, variables y tipos y los operadores que existen.
 - Explicaremos las estructuras de control de flujo, if...else, Switch, For, While...
 - Veremos como crear funciones JavaScript, donde declararlas, como llamarlas, sus argumentos, funciones predefinidas, etc.
 - Descubriremos los objetos, su jerarquía, las propiedades y métodos de los objetos:
 - Del Navegador.
 - Del Documento.
 - Del Lenguaje.
 - Y también nos centraremos en los objetos personalizados.

INTRODUCCIÓN



En esta unidad vamos a descubrir como JavaScript, que es un tipo de lenguaje, permitirá a los desarrolladores crear acciones en sus páginas web. Pero qué es JavaScript y cómo nace JavaScript son algunas de la preguntas que desvelaremos a lo largo de esta unidad.

Veremos que JavaScript es un lenguaje que va a ser utilizado por profesionales y por desarrolladores que se inician en el desarrollo y diseño de sitios web. Descubriremos que el lenguaje funciona del lado del cliente, y no requiere de su compilación, ya que los encargados de interpretar este código son los navegadores.

En esta unidad veremos toda la sintaxis básica de JavaScript y conoceremos los elementos más importantes para entender su arquitectura y funcionamiento.

1.1. ¿Qué es JavaScript?

JavaScript podríamos definirlo no como un lenguaje de programación propiamente dicho, sino como un lenguaje de programación interpretado. Y decimos interpretado, ya que no es necesario compilar los programas para ejecutarlos.

Dicho de otra manera, los programas realizados en *JavaScript* se pueden ejecutar directamente en el navegador sin necesidad de procesos intermedios, siendo utilizado para realizar páginas web dinámicas.



Página web dinámica
Es aquella página que incorpora en ella efectos como textos que aparecen y desaparecen, acciones que se activan al pulsar botones, animaciones y ventanas emergentes con mensajes de aviso al usuario.

A pesar de su nombre, debemos saber que *JavaScript* no guarda ninguna relación directa con el lenguaje de programación *Java*. Hablando en términos legales, *JavaScript* es una marca registrada de la empresa *Oracle* y *Sun Microsystems*. Se puede ampliar esta información en su página oficial:

<http://www.sun.com/suntrademarks/>

1.1.1. Historia

Los comienzos de *JavaScript* se remontan a los años 90, más exactamente al año 1995, cuando Netscape introdujo su versión 2.0 del Navegador, el cual incluía *JavaScript* con el nombre de *Mocha*. Aunque poco después se le denominó *LiveScript*, para acabar denominándose *JavaScript*.

Dicha modificación del nombre coincidió con la versión en la que Netscape comenzó a dar soporte para la tecnología *Java*. Esta modificación hizo pensar que *JavaScript* era una prolongación de *Java*, y realmente fue un movimiento de marketing, ya que se había firmado una alianza entre *Netscape* y *Sun Microsystems*, y este último había lanzado meses antes el lenguaje de programación *Java*.

JavaScript triunfaba en todo el mundo y *Netscape* 3.0 incorporó poco después la versión 1.1 de este lenguaje. *Microsoft*, que vio el éxito de este lenguaje, lanzó una copia de *JavaScript* al que le cambiaron el nombre para evitar problemas legales. Lo denominaron *JScript* y fue lanzado con su versión de navegador Internet Explorer 3.

Netscape decidió evitar una guerra de tecnologías estandarizando el lenguaje *JavaScript*. Así, en 1997 envió la especificación de *JavaScript* 1.1 al *ECMA International*



NOTA

ECMA International

Ecma International es la organización internacional dedicada a realizar estándares para la comunicación y la información. Anteriormente llamada ECMA (*European Computer Manufacturers Association*), en 1994 cambió su nombre para indicar su alcance internacional.

ECMA realizó una “estandarización del lenguaje de script multiplataforma e independiente de cualquier empresa”. Este primer estándar se denominó “ECMA-262”, en el que se definió por primera vez el lenguaje *ECMAScript*. De hecho, *JavaScript* no es más que la implementación que realizó la empresa Netscape del estándar *ECMAScript*.



ECMA-262

Puede consultar toda la información referente a este estándar en la siguiente dirección:

<http://www.ecma-international.org/publications/standards/Ecma-262.htm>

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.

Actualmente, *JavaScript* es una marca registrada de Oracle tras la adquisición por parte de esta de Sun Microsystems.

1.2. ¿Cómo incluir JavaScript en documentos XHTML?

La forma de integrar *XHTML* y *JavaScript* en las páginas web se puede realizar al menos de tres formas diferentes que vamos a ver a continuación.

1.2.1. Introducirlo en el mismo documento XHTML

Debemos saber que el código *JavaScript* se encierra entre las etiquetas `<script> </script>` y se puede incluir en cualquier parte del documento, aunque se recomienda definir el código *JavaScript* dentro de la cabecera del documento, entre las etiquetas `<head> </head>`.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head runat= "server">
    <title> Página sin título </title>
    <script type= "text/javascript">
      "codigo JavaScript"
    </script>
  </head>
  <body>
    </body>
</html>
```

Debemos tener en cuenta que para que la página *XHTML* resultante sea válida, será necesario añadir el atributo `type` a la etiqueta `<script>`. Los valores que incluiremos en el atributo `type` están estandarizados y para el uso de *JavaScript*, el valor que debemos poner es `"text/javascript"`.

Debemos emplear este método cuando definimos un bloque pequeño de código o cuando queremos incluir instrucciones específicas en un determinado documento de *HTML*, completando de esta manera las instrucciones y funciones que por defecto están en todos los documentos de un sitio web. El problema que tenemos de usar esta forma de introducir código *JavaScript*, es que si queremos realizar una modificación, deberemos realizarla en todas las páginas que contengan dicho código *JavaScript*.

1.2.2. Definir JavaScript en un archivo externo

Otra forma de incluir código *JavaScript* en nuestras páginas web, es definirlo en un archivo externo de tipo *JavaScript*, que podemos enlazar con los documentos *XHTML* mediante la etiqueta `<script>`. Podemos crear los archivos *JavaScript* que necesitemos y enlazarlos a nuestro documento *XHTML*.

Por ejemplo, vamos a enlazar el documento `ejemplo.js` donde tenemos definido el código de ejemplo para que sea llamado cuando cargue la página web en nuestro navegador.

Para ello debemos definirlo dentro de las etiquetas `<script>` de la siguiente forma:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html>  
<head>  
  <title>Página sin título</title>  
  <script type="text/javascript" src="/js/ejemplo.js"></script>  
</head>  
<body>  
</body>  
</html>
```

Podemos ver, que aparte del atributo `type` debemos definir también el atributo `src`, que es el que indica la URL o dirección correspondiente al archivo *JavaScript* al que queremos enlazar. Solamente podemos enlazar a un único archivo por cada etiqueta `<script>`, pero podemos insertar tantas etiquetas como sean necesarias.



Archivo.js

Un archivo *JavaScript*, es un archivo de texto plano cuya extensión es `.js`. Éstos archivos se pueden crear con editores de texto como el *Notepad*, *Wordpad*, *Notepad++*, *UltraEdit*, etc

La mayor ventaja de usar archivos *JavaScript* reside en que de esta manera simplificamos el código XHTML de nuestra página web, pudiendo reutilizar el mismo código en todas las páginas de nuestro sitio web, y por lo tanto, cualquier modificación que tengamos que realizar, únicamente será necesario hacerla en dicho archivo y se verá reflejada en todas las páginas XHTML que lo mencionan.

1.2.3. Insertarlo en los elementos XHTML

Esta forma de introducir JavaScript en una página web es la menos utilizada. Consiste en introducir las partes del código dentro del código XHTML de la página.

Veamos un pequeño ejemplo en el que introducimos un mensaje de alerta dentro del código:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html>  
<head>  
    <title>Página ejemplo de código insertado en el XHTML</title>  
</head>  
<body>  
    <p onclick="alert('Mensaje de prueba')">Introduzca el texto aquí</p>  
</body>  
</html>
```



La sentencia `alert` nos muestra en pantalla el mensaje que se le pasa por parámetro y un botón “aceptar”. El mensaje no desaparecerá hasta que el usuario pulse sobre el botón.

Si usamos este método para insertar el código JavaScript, aparte de ensuciar nuestro código, se nos presentan muchos problemas posteriores a la hora de realizar cambios y mejoras de dicho código, puesto que es mucho más costoso tener que revisar todo el código XHTML, que hacerlo en el bloque de la cabecera de la página o realizarlo en el archivo .js.

Los inconvenientes que genera esta forma de insertar código JavaScript hace que sea la forma menos usada.

1.3. La etiqueta noscript

Aunque ya existen pocos, algunos navegadores no soportan el uso del lenguaje JavaScript, y otros permiten bloquearlo. Muchos usuarios piensan que bloqueándolo navegan de una forma más segura.

Para estos casos excepcionales, en los que las páginas web necesitan JavaScript para su correcto visionado, se incluye un aviso mediante un mensaje en el que se avisa al usuario que debe activar *JavaScript* en su navegador para poderla visualizar correctamente.

La forma de hacerlo es mediante la etiqueta <`noscript`>:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>Página Web</title>
</head>
<body>
  <noscript>
    <p>Bienvenidos a mi Página Web</p>
    <p>Esta página necesita tener habilitado JavaScript.
      Activelo si lo ha inhabilitado o cambie de navegador</p>
  </noscript>
</body>
</html>
```



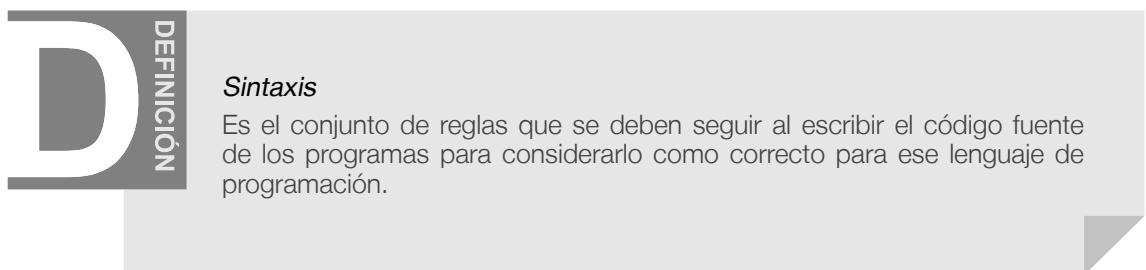
Etiqueta <noscript>

El contenido de la etiqueta <`noscript`> suele mostrarse cuando el navegador del usuario que visita la página no soporta *JavaScript*. Esta etiqueta es muy importante puesto que muestra una alternativa al *JavaScript* que hemos debido incluir en nuestras páginas en el caso de que el visitante no lo soporte.

La etiqueta <`noscript`> debe estar incluida entre las etiquetas <`body`> para que se muestre al usuario cuando se cargue la página.

1.4. Sintaxis

En este punto vamos a definir las normas básicas que definen la sintaxis de *JavaScript*.



Sintaxis
Es el conjunto de reglas que se deben seguir al escribir el código fuente de los programas para considerarlo como correcto para ese lenguaje de programación.

La sintaxis de *JavaScript* es muy similar a la de otros lenguajes de programación. A continuación vamos a definir las normas básicas que definen la sintaxis de *JavaScript*:

- **No tendremos en cuenta los espacios en blanco y las nuevas líneas:** al igual que sucede con XHTML, el intérprete de *JavaScript* va a ignorar cualquier espacio en blanco sobrante, de esta manera podemos ordenar el código de forma que podamos entenderlo mejor (realizando tabulaciones de las líneas, añadiendo espacios, creando nuevas líneas, etc.)
- **Debemos distinguir las mayúsculas y minúsculas:** si en una página XHTML utilizamos indistintamente mayúsculas y minúsculas, la página la vamos a visualizar correctamente, siendo el único problema la no validación de la página. En cambio, si en *JavaScript* intercambiamos las mayúsculas y minúsculas el script no funcionará.
- **No será necesario definir el tipo de las variables:** en *JavaScript* no es necesario indicar el tipo de dato que necesitaremos almacenar cuando creamos una nueva variable. De esta manera, en una misma variable podremos almacenar diferentes tipos de datos mientras se ejecuta el script.
- **Cuando terminemos una sentencia no será necesario insertar un punto y coma (;**: ya sabemos que la mayoría de lenguajes de programación nos obligan a terminar cada sentencia con dicho carácter. Y aunque en *JavaScript* no sea obligatorio el hacerlo, recomendamos terminar cada sentencia con el carácter del punto y coma (;) para que en otros lenguajes no nos cueste el hacerlo.
- **Nos permite insertar comentarios:** con los comentarios añadimos información adicional para nuestra ayuda en el código fuente del programa. Debemos saber que, aunque el contenido de los comentarios no se van a ver por pantalla, sí que va a ser enviado al navegador del usuario junto con el resto del script, por lo que es recomendable extremar las precauciones de los comentarios que vamos a insertar en el código.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>Página Web</title>
</head>
<body>
<script type="text/javascript">
// Para comentarios de una sola linea.

/* Para comentarios en los que queremos insertar mas informacion
y necesitanmos de varias lineaas */

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

1.5. Posibilidades y limitaciones de JavaScript

Hasta la aparición de *Flash*, *JavaScript* fue utilizado de una forma masiva por la mayoría de los sitios de Internet. *Flash* daba la posibilidad de realizar ciertas acciones que resultaba imposible ejecutar en *JavaScript*.

En cambio, con la aparición de las aplicaciones *AJAX* programadas con *JavaScript* se ha conseguido devolver la popularidad a dicho lenguaje de programación web.

Para hablar de las limitaciones de *JavaScript* debemos pensar que fue diseñado para que se ejecutara en un entorno muy limitado, añadiendo la limitación de que los usuarios tenían que confiar en la ejecución de los scripts.

Así, los scripts de *JavaScript* no se pueden comunicar con recursos externos al dominio desde el que se descargó el script. Estos tampoco pueden cerrar ventanas que no hayan sido abiertas por esos mismos scripts. Las ventanas creadas con *JavaScript* no pueden ser ni demasiado pequeñas ni demasiado grandes y además deben estar siempre a la vista del usuario (aunque es cada navegador el que dicta los detalles concretos).

Otra de las limitaciones es que los scripts no van a poder acceder a los archivos del ordenador del usuario y tampoco podrán leer o modificar las preferencias del navegador.

Y ya por último, si durante la ejecución de un script, ya sea por un error en la programación del script o por otros motivos, este tiempo es demasiado largo, el navegador dará la posibilidad de detener su ejecución, no sin antes informar al usuario de que dicho script está consumiendo demasiados recursos.

1.5.1. JavaScript en los navegadores

En la actualidad, la mayoría de los navegadores disponibles incluyen soporte de *JavaScript* hasta la versión correspondiente a la tercera edición del estándar ECMA-262.

La mayor diferencia entre ellos reside en el lenguaje utilizado, Internet Explorer es el único que lo hace utilizando un lenguaje diferente *JScript*, el resto de navegadores como Chrome, Firefox, Opera, Safari, Konqueror, etc., utilizan *JavaScript*.



Lenguaje JScript

Si necesita más información acerca de este lenguaje puede consultar las páginas oficiales de Microsoft:

[http://msdn.microsoft.com/es-es/library/72bd815a\(v=VS.80\).aspx](http://msdn.microsoft.com/es-es/library/72bd815a(v=VS.80).aspx)

1.6. Nuestro primer script

En este punto vamos a realizar el primer script sencillo pero completo, con el que vamos a explicar todos los elementos que lo componen y además veremos como ejecutarlo en los diferentes navegadores para ver la forma en que reacciona en cada uno de ellos.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
  
<html xmlns="http://www.w3.org/1999/xhtml">  
  
<head>  
  
    <meta http-equiv="Content-Type" content="text/html;  
charset=iso-8859-1"/>  
    <title>Practica Nuestro primer script</title>  
    <script type="text/javascript">  
        alert("Seas, Hola Mundo!");  
    </script>  
</head>  
<body>  
    <p>Ejemplo de Nuestro primer Script</p>  
</body>  
  
</html>
```

En este primer ejemplo, hemos incluido el script como un bloque de código dentro de una página XHTML. Para ello, debemos crear una página XHTML correcta que incluya la declaración del **DOCTYPE**, el atributo **xmlns**, las secciones **<head>** y **<body>**, la etiqueta **<title>**, etc.



DOCTYPE

Según los estándares del HTML, cada documento requiere un tipo de declaración del documento. El documento del HTML comienza con el “DOCTYPE” y le dice que versión del HTML utilizar en la comprobación de sintaxis del documento.

Para más información sobre los distintos DOCTYPE, puedes visitar esta dirección:

<http://htmlhelp.com/tools/validator/doctype.html>

Como ya comentamos en capítulos anteriores, recomendamos incluir el bloque código en la cabecera del documento, es decir, dentro de la etiqueta **<head>**. Posteriormente debemos introducir el código **JavaScript** entre las etiquetas **<script></script>**, y para que la página sea válida, tenemos que definir el atributo **type** de la etiqueta **<script>**. El atributo **type** se corresponde con “el tipo **MIME**”.



MIME

Es un estándar para identificar los diferentes tipos de contenidos. El “tipo MIME” correcto para JavaScript es “*text/javascript*”.

Después vamos a escribir todas las sentencias que forman la aplicación. Este primer ejemplo solamente incluye una sentencia: `alert("Seas, Hola Mundo!");`.

La instrucción `alert()` nos permite mostrar un mensaje en la pantalla del usuario. Si visualizamos la página web de este “*Nuestro Primer script*” en cualquier navegador, automáticamente se nos mostrará una ventana con el mensaje “*Seas, Hola Mundo!*”.

A continuación mostramos el resultado de ejecutar el script en diferentes navegadores:

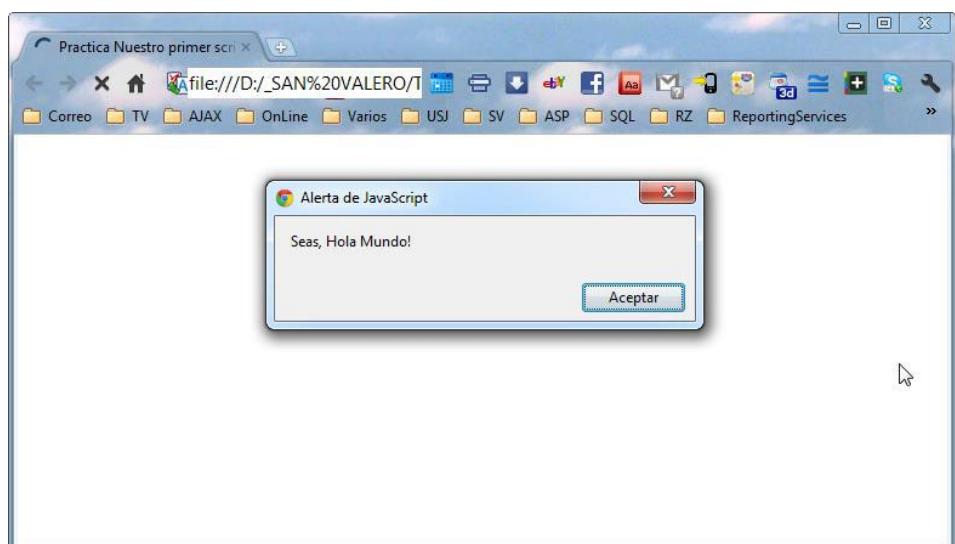


Figura 1.1. Mensaje con `alert()` en Google Chrome.

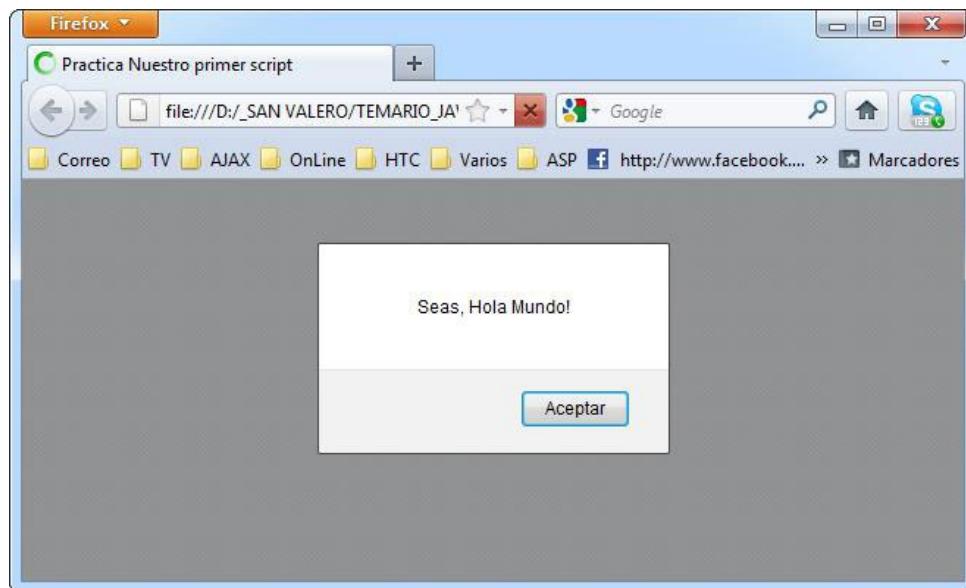


Figura 1.2. Mensaje con `alert()` en Firefox.

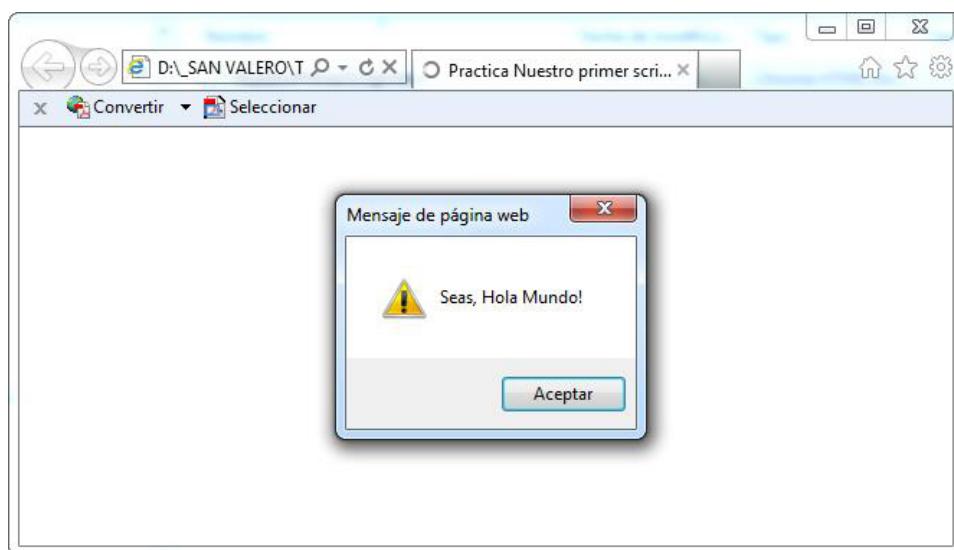


Figura 1.3. Mensaje con `alert()` en Internet Explorer.

Podemos observar que la funcionalidad de `alert()` en los distintos navegadores es la misma, aunque sí que existen diferencias visuales entre ellos.

1.7. Elementos básicos

Este apartado se centra en los elementos básicos que el usuario debe conocer para manejar el programa. Estos elementos están conformados por variables y operadores y ambos se desarrollan a continuación con detalle.

1.7.1. Variables y tipos de variables

Lo primero que tenemos que saber es que las variables en los lenguajes de programación siguen una lógica similar a las variables utilizadas en otros ámbitos como pueden ser las matemáticas.



DEFINICIÓN

Variables

Una variable es un elemento que se emplea para almacenar y hacer referencia a otro valor.

De esta forma, podemos crear programas que funcionan siempre igual independientemente de los valores concretos utilizados. Al igual que en las matemáticas, donde las variables nos sirven para definir las fórmulas y ecuaciones, en programación necesitamos de las mismas para realizar los programas.

Para poner un ejemplo, si quisieramos realizar un programa en el que deseáramos multiplicar dos números, lo siguiente:

```
Resultado = 2 * 3;
```

El programa anterior es tan poco útil que sólo nos serviría para el caso en el que el primer número del producto sea el 2 y el segundo número sea el 3. En cualquier otro caso, el programa obtendría un resultado incorrecto.

En cambio, si realizamos el programa usando variables para almacenar cada uno de los números en cada una de ellas:

```
valor1 = 2;  
valor2 = 3;  
  
Resultado = valor1 * valor2;
```

Los elementos `valor1` y `valor2` son las variables que guardarán los valores que posteriormente utilizará nuestro programa. El `Resultado` variará en función del valor almacenado por las variables, por lo que hemos conseguido que nuestro programa funcione correctamente para cualquier par de valores indicado. Si modificamos el valor de cualquiera de las variables, el programa seguirá funcionando correctamente.

En JavaScript, las variables se crean mediante la palabra reservada `var`.



Palabra Clave Reservada

Son las palabras (en inglés) que se utilizan para construir las sentencias de JavaScript y que, por tanto, no pueden ser utilizadas libremente. Las palabras actualmente reservadas por JavaScript son: `break`, `case`, `catch`, `continue`, `default`, `delete`, `do`, `else`, `finally`, `for`, `function`, `if`, `in`, `instanceof`, `new`, `return`, `switch`, `this`, `throw`, `try`, `typeof`, `var`, `void`, `while`...

Si el programa anterior lo realizamos en JavaScript, quedaría de la siguiente manera:

```
<script type="text/javascript">  
var valor1 = 2;  
var valor2 = 3;  
var Resultado = valor1 * valor2;  
</script>
```

La palabra reservada `var` únicamente se debe indicar cuando se define por primera vez la variable, lo que se denomina “declarar una variable”. En el resto del script solo será necesario indicar su nombre. En caso contrario nos daría un error.



Declarar una variable

Es la definición de un nombre como variable para usarlo posteriormente durante la ejecución de un programa.

Inicializar una variable

Si en el momento de declarar una variable se le asigna también un valor.

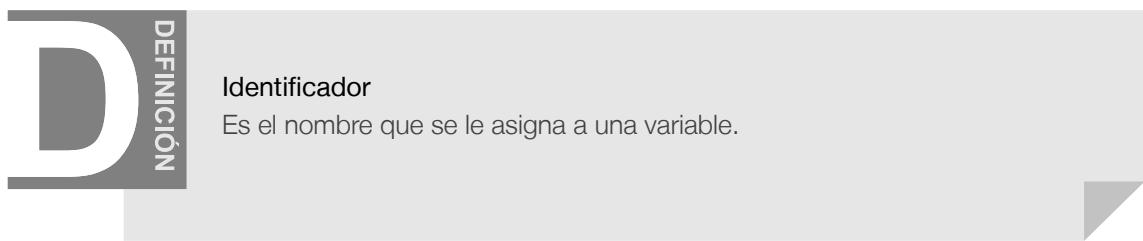
La inicialización de variables en *JavaScript* es opcional, ya que se pueden declarar por una parte y posteriormente asignarles un valor.

De esta manera, podríamos rehacer el ejemplo anterior de la siguiente forma:

```
<script type="text/javascript">
    var valor1;
    var valor2;
    valor1 = 2;
    valor2 = 3;
    var Resultado = valor1 * valor2;

</script>
```

Para un programador habituado a otros lenguajes de programación, el que no sea necesario declarar las variables le puede resultar extraño, por lo que para que cuando se vaya a programar en otros lenguajes en los que sí que sea obligatorio definirlas, recomendamos que siempre se realice la declaración de las mismas.



Los identificadores en *JavaScript* deben cumplir dos normas básicas:

- Sólo pueden estar formados por letras, números y los símbolos de dólar "\$" y guión bajo "_" .
- El primer carácter del identificador no puede ser un número.

Por lo dicho anteriormente, ejemplos válidos de identificadores de variables serían los siguientes:

```
<script type="text/javascript">
    var $valor1;
    var _$letras;
    var $$$cualquierNumero;
    var $_g__$46

</script>
```

Llegados a este punto, en que ya conocemos qué es y cómo se declaran las variables, vamos a ver los diferentes tipos que existen en función del tipo de valor que queramos almacenar.

Numéricas

Estas variables las usaremos para almacenar valores numéricos enteros (*integer*) o decimales (*float*). No es necesario asignar el tipo, ya que se realiza indicando directamente en número entero o decimal al declarar la variable.

```
<script type="text/javascript">  
    var iva = 18;          // variable de tipo entero  
    var euro = 166.386;    // variable de tipo decimal  
</script>
```

Los números decimales utilizan el carácter punto “.” en vez de coma “,” para separar la parte entera y la parte decimal.

Cadena de texto

Este tipo de variables las usaremos para almacenar caracteres, palabras y/o frases de texto. Cuando queramos asignar el valor a la variable, deberemos utilizar las comillas dobles o simples, para delimitar su comienzo y su final:

```
<script type="text/javascript">  
    var texto1 = "Texto encerrado entre comillas dobles";  
    var texto2 = 'Texto encerrado entre comillas simples';  
</script>
```

Si en alguna ocasión necesita guardar un texto que contiene comillas dobles o simples, el truco que se debe utilizar es el de encerrar el texto con las comillas que no utilice el texto:

```
<script type="text/javascript">  
    /* En la variable texto1 insertamos comillas simples, por lo  
    que lo encerramos con comillas dobles */  
    var texto1 = "Texto con 'comillas simples' dentro";  
    /* En la variable texto2 insertamos comillas dobles, por lo que  
    lo encerramos con comillas simples */  
    var texto2 = 'Texto con "comillas dobles" dentro';  
</script>
```

Aparte de las comillas, existen otros caracteres difíciles de incluir en una variable de texto como son el tabulador o el ENTER. Para resolver estos problemas, en JavaScript se sustituye el carácter problemático por una combinación simple de caracteres.

En la tabla siguiente podemos ver la conversión que se debe utilizar:

Texto a incluir	Deberemos incluir
Una nueva línea	\n
Un tabulador	\t
Una comilla simple	\'
Una comilla doble	\\"
Una barra inclinada	\\\

Figura 1.4. Tabla de conversión de caracteres especiales.

Una vez que ya sabemos cómo cambiar los caracteres especiales, el ejemplo anterior podría quedar definido de la siguiente forma:

```
<script type="text/javascript">

    /* En la variable texto1 insertamos comillas simples, por lo
    que lo encerramos con comillas dobles */

    var texto1 = 'Texto con \'comillas simples\' dentro';

    /* En la variable texto2 insertamos comillas dobles, por lo que
    lo encerramos con comillas simples */

    var texto2 = "Texto con \"comillas dobles\" dentro";

</script>
```

En el capítulo de funciones veremos las funciones propias del objeto string.

Arrays

Podemos definir un Array como un conjunto de elementos del mismo tipo colocados de forma ordenada, de manera que nos podremos referir a ellos con un nombre común y un índice de lugar.

D

DEFINICIÓN

Array

Es una colección de variables, que pueden ser todas del mismo tipo o cada una de un tipo diferente, ordenados y a los cuales accedemos a través de su identificador y un índice de posición.

A los Arrays también se les llama vectores o matrices, aunque el término *Array* es el más utilizado y es una palabra más aceptada en el entorno de la programación.

Las matrices pueden ser de una o más dimensiones. Cuando son de una dimensión se llaman *Arrays*. En JavaScript los *Arrays* son todos de UNA sola dimensión. Veamos un gráfico que nos ayudará a saber cómo funciona un *Array*:



Figura 1.5. Ejemplo gráfico de un Array.

La figura anterior muestra un ejemplo gráfico de un *Array*. En cada “caja” se guardaría la información. Un *Array* o *Matriz* de dos dimensiones podríamos representarlo de la siguiente manera:

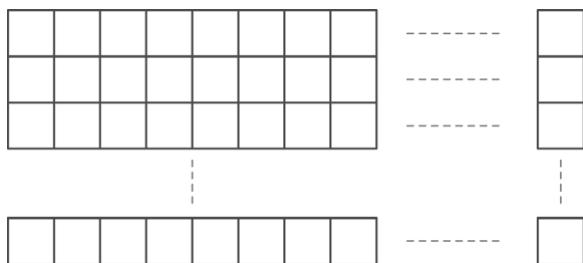


Figura 1.6. Ejemplo gráfico de Array de dimensión 2.

Como esta estructura se utiliza con frecuencia en programación, por ello profundizaremos en sus características y utilización en un capítulo dedicado a los Arrays.

Booleanos

Las variables **booleanas** se denominan también variables de tipo lógico. Su funcionamiento básico es muy sencillo. Almacenan un valor que únicamente varía entre dos valores `true` o verdadero y `false` o falso. En este tipo de variable nunca podremos almacenar ni números ni cadenas de texto.

```
<script type="text/javascript">  
    var altaCliente = true;  
    var ivaIncluido = false;  
</script>
```

Nulas

JavaScript dispone de un valor especial, `null`, que nos indica que la variable no tiene valor. Así, si una variable es `null` es que está vacía. Debemos saber que una variable vacía no ocupa memoria.

```
<script type="text/javascript">  
    var valor = 10;  
    valor = null;  
</script>
```

Conversión entre tipos de datos

Debemos conocer que una variable en *JavaScript* tiene el tipo de datos del dato que contiene. Pero nosotros podemos forzar a una conversión, si en algún momento nos hace falta.

Las tres funciones de conversión que veremos son:

- ***parseInt(string, base)***: devuelve la cadena de texto en número entero en base a la pasada como parámetro. Lo normal será usar la base 10.
- ***parseFloat(string)***: devuelve la cadena de texto en número decimal.
- ***toString()***: devuelve una cadena de texto a partir de un número de entrada.

```
<script type="text/javascript">  
    A = parseFloat("100,05");  
    B = parseInt("100,05", 10);  
    C = A.toString();  
</script>
```

Funciones y propiedades básicas JavaScript

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.

1.7.2. Operadores

En el capítulo anterior hemos conocido las variables. Por si solas vemos que son de poca utilidad. Para poder realizar programas que implementen funcionalidades más avanzadas, necesitamos de los operadores.

Los operadores nos van a permitir manipular el valor de las variables, realizar operaciones matemáticas con sus valores y realizar comparaciones de variables. Los operadores facilitan a los programas poder realizar cálculos complejos y tomar decisiones lógicas dependiendo de las comparaciones y otros tipos de condiciones que se hayan fijado.

1.7.2.1. Operador de asignación

Con este operador guardamos un valor en una variable. Es el más utilizado de todos. Su símbolo es “=”.

```
<script type="text/javascript">  
    var valor1 = 10;  
    var valor2 = valor1 + 10;  
    var valor3 = valor2;  
</script>
```

Fíjate que a la izquierda del operador siempre debemos indicar el nombre de la variable, y a la derecha del mismo pueden ir valores, variables, condiciones lógicas, etc.

1.7.2.2. Operadores de incremento y decremento

Estos dos operadores los vamos a utilizar para incrementar o decrementar en una unidad el valor de una variable. Debemos saber que solamente son válidos para las variables numéricas.

Para usar el operador de incremento debemos usar el prefijo “++” delante del nombre de la variable. Como resultado obtendremos el valor de esa variable incrementado en una unidad.

```
<script type="text/javascript">  
    var valor = 10;  
    ++valor; // valor = 11  
    // sería lo mismo que hacer lo siguiente  
    var valor1 = 10;  
    valor1 = valor1 + 1; // valor1 = 11  
</script>
```

De forma similar a la anterior, debemos usar el prefijo “--” para decrementar el valor de la variable en una unidad. Veamos otro ejemplo:

```
<script type="text/javascript">  
    var valor = 10;  
    --valor; // valor = 9  
    // sería lo mismo que hacer lo siguiente  
    var valor1 = 10;  
    valor1 = valor1 - 1; // valor1 = 9  
</script>
```

Pongamos especial atención ahora. Ambos operadores se pueden usar también como sufijo. Podemos observar su funcionamiento en el siguiente ejemplo:

```
<script type="text/javascript">  
    var valor = 10;  
    valor++; // valor = 11  
</script>
```

El resultado en este caso es el mismo usando el operador como prefijo que como sufijo, pero realmente funcionan de forma muy diferente. Con el siguiente ejemplo podrás ver las diferencias entre usarlo como prefijo o como sufijo:

```
<script type="text/javascript">  
    var valor1 = 5;  
    var valor2 = 2;  
    valor3 = valor1++ + valor2;  
    // valor3 = 7, valor1 = 6  
    var numero1 = 5;  
    var numero2 = 2;  
    numero3 = ++numero1 + numero2;  
    // numero3 = 8, numero1 = 6  
</script>
```

Si vemos el ejemplo anterior, los resultados obtenidos por las variables `valor3` y `numero3` son totalmente diferentes.



Operadores de incremento y decremento

Cuando los usamos como prefijo de la variable, su valor se incrementa antes de realizar cualquier otra operación. Si los utilizamos como sufijo, su valor se incrementa después de ejecutar la sentencia en la que aparece.

Por lo tanto debemos pensar antes de usarlo como nos interesa hacerlo para que después no pueda dar resultados confusos y erróneos.

1.7.2.3. Operadores lógicos

Estos operadores lógicos son imprescindibles para poder realizar aplicaciones complejas, ya que los utilizaremos para tomar decisiones sobre qué instrucciones deberá ejecutar el programa en función de ciertas condiciones que definamos con ellos.

Siempre obtendremos un valor lógico o **booleano** de cualquier operador usado en nuestro código. Los operadores lógicos son los siguientes:

"!" - Negación

Este operador lo utilizaremos para obtener el valor contrario al valor de la variable. La negación lógica la obtendremos poniendo el símbolo “!” delante de la variable.

```
<script type="text/javascript">  
    var valor1 = true;  
    var valor2 = false;  
    var valor3 = !valor1; // valor3 = false  
    var valor4 = !valor2; // valor4 = true  
</script>
```

El ejemplo anterior únicamente vale para valores **booleanos**. Si tenemos números o cadenas de texto, necesitamos realizar previamente una conversión.

- Si contiene un número, “0” se convierte en `false`, y cualquier otro número positivo o negativo, decimal o entero se convertiría en `true`.
- Si la variable contiene una cadena de texto sería `false` si la cadena está vacía (“”), y en `true` en cualquier otro caso.

Veamos un ejemplo de como funciona este operador y la conversión anteriormente explicada:

```
<script type="text/javascript">  
    var numero = 0;  
    vacio = !numero; // vacio = true  
    numero = 2;  
    vacio = !numero; // vacio = false  
    var mensaje = "";  
    mensajeVacio = !mensaje; // mensajeVacio = true  
    mensaje = "SEAS";  
    mensajeVacio = !mensaje; // mensajeVacio = false  
</script>
```

"&&" – And

Para poder obtener el resultado de la operación lógica AND debemos combinar dos valores booleanos. El símbolo del operador es “&&” y su resultado `true` solo si los dos operadores de la operación son `true`.

Para ver mejor las combinaciones véase la siguiente tabla:

Variable1	Variable2	Variable1 && Variable2
True	False	False
False	True	False
True	True	True
False	False	False

Figura 1.7. Tabla resultados operador &&.

Para comprender mejor puedes ver el siguiente ejemplo de código:

```
<script type="text/javascript">
    var valor1 = true;
    var valor2 = false;
    resultado = valor1 && valor2; // resultado = false
    valor1 = true;
    valor2 = true;
    resultado = valor1 && valor2; // resultado = true
</script>
```

"||" – OR

Este operador lógico también combina dos valores booleanos. Su representación gráfica la realizaremos mediante el símbolo “||” y su resultado será `true` cuando alguna de las dos condiciones sea `true`.

Variable1	Variable2	Variable1 && Variable2
True	False	True
False	True	True
True	True	True
False	False	False

Figura 1.8. Tabla resultados operador ||.

Podemos ver el siguiente ejemplo de código para entenderlo mejor:

```
<script type="text/javascript">  
    var valor1 = true;  
    var valor2 = false;  
    resultado = valor1 || valor2; // resultado = true  
    valor1 = false;  
    valor2 = false;  
    resultado = valor1 || valor2; // resultado = false  
</script>
```

1.7.2.4. Operadores matemáticos

En JavaScript existen cuatro operadores definidos con los que podemos realizar cálculos matemáticos sobre los valores de las variables numéricas.

Los operadores definidos son:

- Suma “+”
- Resta “-”
- Multiplicación “*”
- División “/”

Veamos un ejemplo con cada uno de ellos:

```
<script type="text/javascript">  
    var valor1 = 6;  
    var valor2 = 2;  
    // suma  
    resultado = 3 + numero1;    // resultado = 9  
    // resta  
    resultado = numero2 - 1;    // resultado = 1  
    // multiplicación  
    resultado = numero1 / numero2; // resultado = 3  
    // división  
    resultado = numero1 * numero2; // resultado = 12  
</script>
```

Aparte de estos cuatro operadores básicos, *JavaScript* define el operador “módulo”, que nos calcula el resto de la división entera de dos números. Por ejemplo, si se divide 8 entre 4, la división es exacta y nos da un resultado de 2. El resto de esa división es 0, por lo que el módulo de 8 entre 4 es igual a 0.

En cambio, si se divide 11 entre 5, la división no es exacta, el resultado nos da 2 y el resto 1, por lo que módulo es igual a 1.

El operador módulo en *JavaScript* lo definimos con el símbolo “%”, que no debe confundirse con el cálculo del porcentaje.

```
<script type="text/javascript">

    var valor1 = 8;
    var valor2 = 4;
    var resultado = numero1 % numero2; // resultado = 0
    valor1 = 11;
    valor2 = 5;
    resultado = numero1 % numero2; // resultado = 1

</script>
```

Todos los operadores matemático, los podemos combinar con el operador de asignación “=” y de esta manera abreviar su notación:

```
<script type="text/javascript">

    var valor1 = 9;
    // suma
    valor1 += 3; // valor1 = valor1 + 3 = 12
    // resta
    valor1 -= 1; // valor1 = valor1 - 1 = 8
    // multiplicación
    valor1 *= 2; // valor1 = valor1 * 2 = 16
    // división
    valor1 /= 9; // valor1 = valor1 / 9 = 1
    //módulo
    valor1 %= 4; // valor1 = valor1 % 4 = 1

</script>
```

1.7.2.5. Operadores relacionales

Los operadores relacionales son símbolos que se usan para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es `true`, en caso contrario es `false`. Los operadores relacionales definidos por *JavaScript* son los siguientes:

Símbolo	Nombre	Ejemplo	Significado
<	menor que	a<b	a es menor que b
>	mayor que	a>b	a es mayor que b
==	igual a	a==b	a es igual a b
!=	distinto de	a!=b	a es distinto de b
<=	menor o igual que	a<=5	a es menor o igual que b
>=	mayor o igual que	a>=b	a es mayor o igual que b

Figura 1.9. Símbolos relacionales.

Veamos cada uno de ellos con un ejemplo de código:

```
<script type="text/javascript">
    var valor1 = 5;
    var valor2 = 9;
    // mayor que
    resultado = numero1 > numero2; // resultado = false
    // menor que
    resultado = numero1 < numero2; // resultado = true
    numero1 = 6;
    numero2 = 6;
    // mayor o igual que
    resultado = numero1 >= numero2; // resultado = true
    // menor o igual que
    resultado = numero1 <= numero2; // resultado = true
    // igual que
    resultado = numero1 == numero2; // resultado = true
    // distinto de
    resultado = numero1 != numero2; // resultado = false
</script>
```



Los operadores relacionales no solamente sirven para comparar números, sino que también pueden usarse para comparar cadenas de texto.

En este punto debemos saber que cuando usemos los operadores “>” y “<” para comparar cadenas de texto, siguen la siguiente metodología: cuando comparan las cadenas de texto, comparan letra a letra empezando desde la izquierda hasta que se encuentre una diferencia entre las dos cadenas de texto.

Una vez que han encontrado dos caracteres diferentes, determinan si una letra es mayor o menor que otra de la siguiente manera:

- Las mayúsculas son menores que las minúsculas.
- Las primeras letras del alfabeto son menores que las últimas.

Veámoslo mejor con un ejemplo de código:

```
<script type="text/javascript">
    var texto1 = "abcd";
    var texto2 = "abcd";
    var texto3 = "abcc";
    resultado = texto1 == texto3; // resultado = false
    resultado = texto1 != texto2; // resultado = false
    resultado = texto3 >= texto2; // resultado = false
</script>
```

El primer y segundo caso no necesita explicación alguna. En el tercer caso, empieza comparando de izquierda a derecha la cadena de texto. Cuando llega al 4º carácter de la variable texto3 “c” está antes que “d” en el alfabeto y por lo tanto es menor, con lo que la comparación de que texto3 sea mayor o igual que texto2 es `false`.

1.8. Estructuras de control de flujo

Hasta ahora, los scripts que hemos visto han sido sencillos y lineales, es decir, simplemente se van ejecutando las sentencias simples, una detrás de otra desde el inicio hasta el final.

Sin embargo, con lo que hemos aprendido hasta este momento no podemos realizar programas que por ejemplo, hagan cosas distintas dependiendo del estado de nuestras variables o realicen un mismo proceso muchas veces sin tener que repetir dichas líneas de código una y otra vez.

Para que podamos realizar cosas más complejas en nuestros scripts, vamos a utilizar las estructuras de control de flujo. Con ellas podremos realizar tomas de decisiones y bucles. Para ello vamos a describir en este capítulo varias estructuras.

1.8.1. Estructura If, If...Else

Es la estructura más utilizada en *JavaScript* y en la mayoría de lenguajes de programación. Se usa para realizar una toma de decisiones y es un condicional que se utiliza para realizar una u otra acción en función de una expresión.

Esta estructura funciona de la siguiente forma, primero se evalúa una expresión, si su resultado es `true`, se realizan las acciones relacionadas con el caso positivo.

Veamos el diagrama de flujo de *If, If... else*

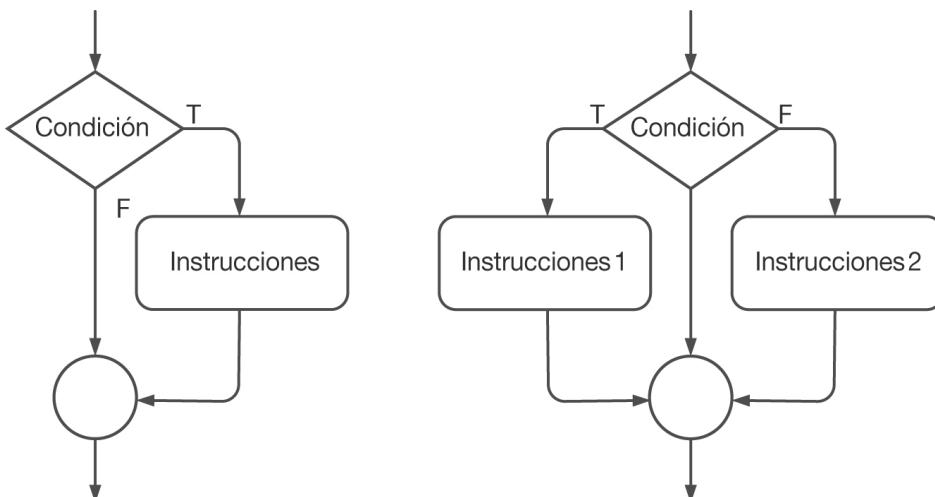


Figura 1.10. Diagramas de flujo de if, if... else.

Como opción podemos definir una acción a realizar en caso de que el resultado de la expresión sea `false`.

Su definición formal es:

```
<script type="text/javascript">
    if (expresión) {
        //acciones a realizar en caso true
        //...
    } else {
        //acciones a realizar en caso false
        //...
    }
</script>
```

Si no definimos la cláusula `else`, el script seguirá ejecutándose hasta el final sin realizar ninguna acción en caso que la expresión sea `false`.

Otra cosa que debemos tener en cuenta es que las llaves, que engloban las sentencias, son opcionales cuando queremos ejecutar una única sentencia. Pero recomendamos utilizarlas siempre para tener un código limpio y claro.

```
<script type="text/javascript">
    if (condicion)
        alert("hacer algo");
    // es lo mismo que poner este código:
    if (condicion){
        alert("hacer algo");
    }
    // e incluso, lo mismo que este otro:
    if (condicion) alert("hacer algo")
</script>
```

La condición que se inserta en el `if(condicion)` puede combinar cualquiera de los diferentes operadores lógicos y relacionales que hemos estudiado en capítulos anteriores:

```
<script type="text/javascript">
    var valor = false;
    if(!valor) {
        alert("Se mostrará este mensaje si \'valor\' es true");
    }
    var valor2 = false;
    var valor3 = true;
    if(!valor2 && valor3) {
```

```
        alert("Solo se mostrará este mensaje si \'valor2\' " +
              "es true y \'valor3\' es true");
    }

var numero1 = 8;
var numero2 = 3;
if (numero1 > 5 || numero2 == 0) {
    alert("Solo se mostrará este mensaje si \'numero1\' " +
          "es mayor que 5 o \'numero2\' es igual a 0");
}

</script>
```

Sentencias anidadas

Las sentencias anidadas las usaremos para poder realizar estructuras condicionales más complejas. De esta manera podremos evaluar y realizar más acciones que únicamente las que nos ofrece una sola. Anidar sentencias consiste en colocar estructuras IF dentro de otras estructuras IF, de esta manera crearemos un flujo de código necesario para decidir correctamente.

Esto nos puede servir, por ejemplo, cuando tenemos que evaluar tres posibilidades distintas. Lo vamos a ver gráficamente con un ejemplo. Imaginemos que queremos saber quién de dos personas es más mayor sabiendo sus edades. Para saber quién es mayor, debemos comprobar si no tienen la misma edad y cuál es mayor o menor.

```
<script type="text/javascript">
    var paco=30;
    var pepe=33;
    // primero comprobamos si tienen la misma edad
    if (paco == pepe) {
        document.write("Paco y Pepe tienen la misma edad")
    }
    else{
        if (paco > pepe) {
            document.write("Paco es mayor que Pepe")
        }
        else{
            document.write("Pepe es mayor que Paco")
        }
    }
</script>
```

Veamos el flujo del programa. Primero evaluamos si paco y pepe tienen la misma edad. Si es así, mostraremos un mensaje informando de ello. Si no lo son, debemos averiguar cuál de los dos es mayor. Para eso debemos hacer otra comparación para saber si paco es mayor que pepe.

Si en esta comparación nos da `true` mostramos un mensaje diciendo que “Paco es mayor que Pepe”, en caso contrario indicaremos que “Pepe es mayor que Paco”.

1.8.2. Estructura switch

En este punto vamos a estudiar la estructura de control “switch”, con la que podremos realizar múltiples operaciones dependiendo del estado de una variable. La utilizaremos cuando tengamos múltiples posibilidades de resultado en la evaluación de una sentencia.

```
<script type="text/javascript">
    switch (expresión a evaluar) {
        case valor1:
            // Sentencias a ejecutar
            break
        case valor2:
            // Sentencias a ejecutar
            break
        case valor3:
            // Sentencias a ejecutar
            break
        default:
            /* Sentencias a ejecutar si no
               cumple ninguna de las anteriores */
    }
</script>
```

Lo primero se evalúa la expresión, dependiendo del valor que resulte, se ejecutarán las sentencias que hayan definido en cada caso. Si ninguno de los valores coincide con el de la expresión evaluada anteriormente, se ejecutan las sentencias definidas en el caso `default`.

Si observamos el código, hemos insertado la palabra clave `break`. Con ella lo que hacemos es que cuando uno de los valores coincide con alguno de los definidos dentro del `switch`, haremos un salto hasta el fin del mismo, sin pasar por las siguientes sentencias.

Si no la ponemos desde el momento que se encuentre coincidencia con uno de los valores, se seguirá pasando por todos los `case` evaluándose cada uno de ellos y ejecutándose los que coincidan con el valor. También es opcional la opción `default` u opción por defecto.

EJEMPLO

Para entenderlo mejor, vamos a realizar un ejemplo práctico implementando el *switch*.

No sé si sabrás que el cálculo de la letra del Documento Nacional de Identidad (DNI) es un proceso matemático sencillo que se basa en obtener el resto de la división entera del número de DNI y el número 23. A partir del resto de la división,

Para calcular la letra de DNI, se realiza mediante un proceso matemático sencillo. Debemos obtener el resto de la división entera del número del DNI y el número 23. Con el resto de esa división obtenemos la letra seleccionándola del Array de letras siguiente:

```
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X', 'B',  
'N', 'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];
```

Si el resto de la división es 0, la letra del DNI es la 'T' y si el resto es 4 la letra es la 'G'.

Sabiendo esto, vamos a realizar un pequeño script que:

- Vamos a guardar en una variable el número de DNI y en otra variable la letra del DNI que dará el usuario.
- Lo primero comprobaremos si el número está comprendido entre 0 y 999999999. Eso significará que el usuario ha introducido mal el número y se lo mostraremos con un mensaje.
- Si el número es válido, calcularemos la letra del Array que hemos dado antes.
- Cuando ya esté calculada, la compararemos con la que hemos almacenado en el primer punto. Si coinciden, mostraremos un mensaje como que son correctos los campos introducidos, y si no lo es, mostraremos un mensaje como que la letra introducida es incorrecta.

Veamos la solución del ejemplo:

```
<script type="text/javascript">  
  
var letras = ['T', 'R', 'W', 'A', 'G', 'M', 'Y', 'F', 'P', 'D', 'X', 'B', 'N',  
'J', 'Z', 'S', 'Q', 'V', 'H', 'L', 'C', 'K', 'E', 'T'];  
  
// con prompt solicitamos que el usuario introduzca un dato  
  
var numero = prompt("Introduce el número de DNI (sin la letra)");  
  
var letra = prompt("Introduce la letra de tu DNI (en mayúsculas)");  
  
//convertimos la letra en mayusculas por si la ha introducido  
en minusculas
```

```
letra = letra.toUpperCase();

// comprobamos que el numero introducido esté entre 0 y
99999999

if(numero < 0 || numero > 99999999) {

    //si no lo esta sacamos un mensaje y no hace nada mas el programa
    alert("El número introducido no es válido");
}

else {
    /*calculamo el módulo del numero introducido y
       devolvemos la letra del array según el indice del mismo
    */
    var letraCalculada = letras[numero % 23];
    // la comparamos con la letra introducida por el usuario
    if(letraCalculada != letra) {
        // si no coincide sacamos una alerta
        alert("La letra o el número no son correctos");
    }
    else {
        // mostramos mensaje de que son correctos
        alert("El número de DNI y su letra son correctos");
    }
}

</script>
```



La sentencia **prompt** la usaremos para pedir datos al usuario por el teclado y se guardarán en una variable.

1.8.3. Estructura For

En este punto vamos a estudiar una de las estructuras de control con las que podemos producir bucles o repeticiones de una forma muy sencilla. La definición formal de un bucle `for` es la siguiente:

```
for (varControl=valorInicial; condición; expresionIncremento)
{
    //sentencias a ejecutar en cada iteración
}
```

La idea de un bucle `for` es que, mientras la condición indicada se siga cumpliendo, se siga repitiendo la ejecución de las instrucciones definidas dentro del `for`. Y que, después de cada repetición, se actualice el valor de las variables que se utilizan en la condición.

Vamos a explicar cada una de las partes incluidas entre los paréntesis. La `varControl`, será una variable interna o de control del bucle (no hará falta declararla previamente) que la utilizaremos como contador del número de repeticiones del bucle en cada momento. `valorInicial` será el valor inicial que tomará la `varControl`.

La `condición` es la segunda parte que será evaluada cada vez que se realice una iteración del bucle. Es la que contiene la expresión que se deberá cumplir para seguir ejecutando el bucle.

Y la tercera y última parte es la `expresionIncremento`, una expresión que modificará el valor de la `varControl`. Normalmente se trata de una simple suma pero puede ser cualquier expresión que permita en algún momento la salida del programa.

Dentro de las llaves “{ }” insertaremos todas las sentencias que queremos que se ejecuten en cada iteración del bucle.

Veamos un ejemplo sencillo:

```
<script type="text/javascript">
    // Ejemplo de estructura for
    for (i = 1; i <= 4; i++)
    {
        document.write(i + "★");
    }
</script>
```

En este ejemplo observamos que:

- La varControl es i y su valorIncial es 1.
- La condición de continuación es que i sea menor o igual que cuatro.
- La expresionIncremento es i = i ++.
- El bucle, por tanto, se ejecutará 4 veces.

El resultado de la ejecución del programa será: 1*2*3*4.

Vamos a ver otro ejemplo diferente en el que usaremos el `for`. Vamos a realizar un programa que leerá una cadena de caracteres del teclado y la escribirá al revés:

```
<script type="text/javascript">

    // texto es la cadena que se leerá.

    var texto = "";

    // textoReves contendrá la cadena al revés.

    var textoReves = "";

    // guardamos la longitud de la cadena.

    var longitud = 0;

    //pedimos que se introduzca un texto y lo guardamos en la va-
    riable

    texto = prompt("Introduce una cadena:", " ");

    /* medimos la longitud de los caracteres
       de la variable y la asignamos a Longitud*/
    longitud = texto.length;

    /* asignamos la longitud a i, le decimos que pare el bucle
       cuando llegue a 0, y le vamos restando unidades a i */

    for (i = longitud -1; i >= 0; i = i - 1)

    {

        // formamos la cadena con los caracteres
        // de texto leidos del final al principio

        textoReves = textoReves + texto.charAt(i);

    }

    //presentamos la palabra al revés

    document.write(textoReves);

</script>
```



NOTA

Con la sentencia `text.length` calcularemos la longitud en caracteres de una variable de texto introducido.

Con la sentencia `charAt(i)` vemos el carácter determinado con índice `i` de una cadena de texto introducida.

1.8.4. Estructura While

Existen ocasiones en las que no sabemos o no nos va a interesar calcular el número de iteraciones que debe de tener un bucle. Para esas ocasiones existe el bucle `while`.

```
<script type="text/javascript">
    while (condición)
    {
        // sentencias a ejecutar;
    }
</script>
```

La estructura `while` ejecuta un bloque de instrucciones y repite dicha ejecución mientras que se cumpla una condición. Veamos su diagrama de flujo:

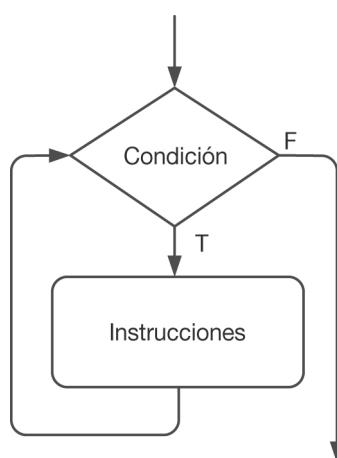


Figura 1.11. Diagrama de flujo de While.

Donde:

- `condición` es la condición cuyo valor deberá ser `true` para que se produzca la entrada en el bucle y que será comprobado antes de cada nueva ejecución del bloque de instrucciones.
- `instrucción` es el bloque de instrucciones que se ejecutará.

Su funcionamiento es el siguiente: el script, cuando encuentre la estructura `while`, antes de entrar por primera vez en el bucle, evaluará la condición. Si esta es `true` entrará en el bucle y se ejecutará el bloque de instrucciones. En cambio, si es `false` ni siquiera entrará en dicho bucle.

Una vez ejecutadas las instrucciones del bucle, se evaluará de nuevo la condición para así poder determinar si se vuelve a ejecutar el bloque o no. Si es `true` se ejecuta, si es `false`, no. Y este punto se repetirá hasta que la condición deje de ser `true`.

Por último, cuando al evaluar la condición el resultado es `false`, el flujo del programa va a la línea siguiente al final del bucle.

Vamos a ver un ejemplo de implementación de la instrucción `while`:

```
<script type="text/javascript">

    /* Declaramos la variable nota y la
       inicializamos en -1 para que entre
       al bucle la primera vez */
    var Nota = -1;

    while (Nota < 0 || Nota > 10)
    {
        Nota = prompt("Introduzca la nota del alumno: ", 0);
        /* con la función Math.round
           redondeamos el valor decimal
           al entero más próximo */

        Nota = Math.round(Nota)
        if (Nota < 0 || Nota > 10)
        {
            alert("Nota erronea")
        }
    }

    // A partir de aquí podemos procesar la nota ya
    // comprobada.

    alert("La nota introducida " + Nota + " es correcta");

</script>
```



Con la sentencia `Math.round` redondeamos el valor decimal al entero más próximo.

En este bucle `while` nos hemos asegurado que se produzca la salida del bucle mediante la condición que hemos definido. Si la condición no la declarásemos de esta manera, el bucle no finalizaría nunca.

También debemos saber que en cualquier momento podemos salir de un bucle mediante la sentencia `break`. Esta sentencia es la misma que vimos cuando estudiamos la sentencia `switch`. Hay que decir que cuando se ejecuta dicha sentencia, se sale inmediatamente del bucle, es decir, que nada de lo que se encuentre debajo del `break` se ejecutará. No es lo normal usarla, pero si lo hacemos, lo deberemos realizar en combinación de `if`, para así definir la condición de salida.



Con la sentencia `break` detendremos la ejecución de un bucle y saldremos de él.

1.8.5. Estructura Do...While

La estructura `do... while` ejecuta el bucle la primera sin comprobar la condición. Para las demás iteraciones del bucle el funcionamiento es idéntico al `while`.

```
<script type="text/javascript">
    do
    {
        // sentencias a ejecutar;
    } while (condición);
</script>
```

Veamos su diagrama de flujo:

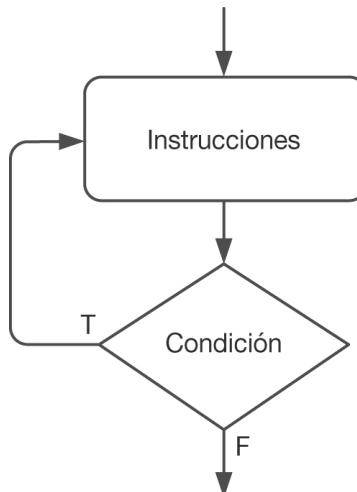


Figura 1.12. Diagrama de Flujo de `do...while`.

A continuación veremos el ejemplo del punto anterior, pero usando un bucle `do...while`.

```

<script type="text/javascript">
  var Nota; // no inicializamos Nota ya que
             // siempre entra en el bucle la primera vez

  do
  {
    Nota = prompt("Introduzca la nota del alumno: ", 0);
    Nota = Math.round(Nota)
    if (Nota < 0 || Nota > 10)
    {
      alert("Nota erronea")
    }
  }
  while (Nota < 0 || Nota > 10)

  alert("La nota introducida " + Nota + " es correcta");
</script>
  
```

Como este bucle se ejecuta siempre, la primera vez no necesitamos inicializar la variable nota con ningún valor. Una vez que ha entrado en el bucle, nos mostrará una ventana solicitándonos que introduzcamos una nota. Mientras esta esté entre los límites de la condición, el bucle seguirá ejecutándose.

Debemos explicar también el posible uso de la sentencia `continue`. Lo usaremos para volver al principio del bucle en cualquier momento, obviando la ejecución de las líneas que haya por debajo de la palabra `continue`.



Con la sentencia `continue` podremos detener la iteración actual del bucle que se esté ejecutando y podremos volver al principio del bucle para realizar otra iteración diferente.

Veamos un ejemplo donde implementamos la sentencia `continue`:

```
<script type="text/javascript">
var num=0
while (num<7)
{
    sumando = prompt("El numero está en " + num + ", dime
si incremento", "Y")
    if (sumando == "N")
        continue
    num++
}
</script>
```

En este caso, si no hubiésemos insertado la sentencia `continue`, simplemente contaría desde `num=0` hasta `num<7`, pero al insertar la sentencia, cada vez que se ejecuta el bucle va a preguntar al usuario si desea incrementar la variable o no. Si introduce “N” se ejecuta la sentencia `continue`, por lo que volvemos al principio del bucle ignorando las sentencias que hayan por debajo el sin incrementar en uno la variable `num`.

1.8.6. Estructura For... In

Aunque podríamos considerarlo como una modificación del bucle `for`, este nos facilita el procesamiento de matrices y colecciones de objetos. La sintaxis de este bloque sería la siguiente:

```
<script type="text/javascript">
for (variable in [objeto|array] )
{
    //instrucciones;
}
</script>
```

Dónde:

- `variable`: es la variable contador, la usaremos para repetir sobre los elementos del array o las propiedades de un objeto.

- [objeto|array]: es el objeto o array del que deseamos conocer sus elementos/propiedades.
- **instrucciones**: son el conjunto de instrucciones que se repetirán.

Al igual que en un bucle `for` normal, utilizamos una variable para iterar, en el caso de que se esté trabajando con un objeto la variable contador del bucle `for` se sustituye por una variable de tipo cadena que contiene el nombre de cada una de las propiedades del objeto manejado en el bucle `for... in`. Si se está trabajando con arrays la variable será de carácter numérico y contendrá la posición del elemento en el array.

En este bucle no es necesario que definamos la condición de terminación, ya que las instrucciones se ejecutarán siempre que existan propiedades del objeto o elementos en el array. Podemos ver que tampoco es necesario incrementar la variable contador ya que se actualiza automáticamente.

Veamos un ejemplo de este bucle:

```
<script type="text/javascript">
    // definimos dos arrays
    var Alumnos = new Array ("MIGUEL", "ELENA", "EVA",
                           "JOAQUIN", "IGNACIO", "ALICIA");
    var Apellidos = new Array ("GIL", "SANCHEZ", "RODRIGUEZ", "GONZALEZ",
                           "CASAUS", "REY");
    for(i in Alumnos)
    {
        document.write(Alumnos[i] + " ** ");
        document.write(Apellidos[i] + "<br>");
    }
</script>
```

1.8.7. Utilización de Arrays mediante estructuras de control

Anteriormente habíamos definido un Array como una serie de elementos, todos del mismo tipo, que ocupan posiciones contiguas en la memoria del ordenador. A estos elementos vamos a acceder mediante un nombre o identificador común para todos los elementos que identifica el Array, y un número o índice que hace referencia al elemento del Array.

A continuación veremos una representación gráfica de un Array que contiene los nombres de 6 amigos:

EVA	ANDRES	MIGUEL	JAVIER	JOSE	CARMEN
Amigos[0]	Amigos[1]	Amigos[2]	Amigos[3]	Amigos[4]	Amigos[5]

Podemos observar que hay seis elementos de información (el Array tiene seis elementos). Todos ellos comparten un mismo identificador (Amigos) pero cada uno tiene, además, un índice que hace referencia al elemento en particular. Podemos ver también que el primer elemento es el 0, el segundo el 1, y así sucesivamente. Así, cuando hagamos referencia a Amigos[4] estaremos refiriéndonos al elemento que ocupa la posición 5 cuyo contenido en este caso es “JOSÉ”.

Vamos a continuar explicando más operaciones con los Arrays en los siguientes puntos.

1.8.7.1. Creación de un Array

Para crear un *Array* utilizaremos el siguiente formato:

```
var NombreArray = new Array (NumeroElementos);
```

Dónde:

- `NombreArray` es el nombre o identificador del *Array*.
- `NumeroElementos` es un número que indica el número de elementos que contendrá.

Por ejemplo, para crear un *Array* que contenga 6 elementos, escribiremos:

```
var array = new Array (6);
```

De esta manera, tenemos creada la estructura y las posiciones de memoria están reservadas y disponibles, aunque vacías porque todavía no hemos introducido en ellas ningún valor.

1.8.7.2. Manipulación de los elementos de un Array

Una vez creada la estructura podemos manipular los elementos del *Array* como si se tratase de variables (con la particularidad del índice) para introducir, cambiar o consultar los valores que contienen.

Para introducir información en un elemento utilizaremos normalmente un operador de asignación. Por ejemplo, para introducir el valor “MIGUEL” en un *Array* que llamaremos `Alumnos` escribiremos: `Alumnos[0] = “MIGUEL”;`

De manera similar introduciremos todos los elementos del *Array*:

```
var Alumnos = new Array (6);  
Alumnos[0] = “MIGUEL”;  
Alumnos[1] = “ELENA”;  
Alumnos[2] = “JORGE”;  
Alumnos[3] = “CRISTINA”;  
Alumnos[4] = “IGNACIO”;
```

```
Alumnos[5] = "ALICIA";
```

Podemos hacer referencia a cualquiera de los objetos del Array, ya sea para cambiar su valor o para utilizarlo en expresiones como si se tratase de cualquier otra variable, con la particularidad apuntada de la utilización del índice. Así, podemos escribir las siguientes expresiones:

- Para visualizar una ventana de alerta con el contenido de `Alumnos[4]`

```
alert(Alumnos[4]);
```

- Para leer un nuevo nombre de la consola e introducirlo en el elemento `Alumnos[4]`

```
Alumnos[4] = prompt("Nombre del alumno", " ");
```

- Para crear la variable `DosPrimeros` e introducir en ella el resultado de concatenar `Alumnos[0] + Alumnos[1]`

```
var DosPrimeros = Alumnos[0] + Alumnos[1];
```

- Para realizar una comparación el contenido de ambos elementos y determinar si el primero es mayor que el segundo según su valor ASCII en cuyo caso el resultado será `true`, en caso contrario `false`.

```
Alumnos[2] > Alumnos[3]
```

En general, para hacer referencia a un elemento de un Array emplearemos el formato genérico:

```
NombreArray[indice]
```

El `indice` no tiene que ser necesariamente una constante numérica. Se puede referenciar mediante cualquier expresión numérica que devuelva un entero. Así, por ejemplo, las siguientes expresiones serían expresiones válidas (suponiendo que `i` sea una variable numérica entera):

- `Alumnos[i]`; hace referencia al elemento `i` (si `i` vale 2, al dos, etcétera).
- `Alumnos[i+1]`; hace referencia al elemento siguiente al elemento `i` (si `i` vale 2, hará referencia el elemento 3).
- `Alumnos[i] > Alumnos[i+1]`; compara el elemento `i` con el siguiente.

1.8.7.3. Recorrido de los elementos de un Array

Las variables numéricas enteras, en combinación con estructuras de control repetitivas, se utilizan frecuentemente para recorrer los elementos de un Array.

Por ejemplo, supongamos la siguiente estructura:

```
for (i = 0; i < 6; i = i + 1)
{
    document.write(Alumnos[i]);
}
```

El bucle `for` se ejecutará 6 veces con la instrucción `document.write()`. Para cada una de estas ejecuciones la variable `i` tomará uno de los valores correspondientes al índice del Array (0, 1, 2, 3, 4 y 5).

En los Arrays está disponible la propiedad `length` que devuelve el número de elementos que tiene un Array (incluyendo los elementos vacíos, si los hubiese). Su formato genérico es:

```
NombreArray.length;
```

De esta manera, la condición del `for`, podríamos escribirla de esta manera

```
for (i = 0; i < NombreArray.length; i = i + 1)
{
    Tratamiento_del_elemento_NombreArray[i];
}
```

Veamos un ejemplo completo de creación del Array, inicialización de los elementos e impresión de los mismos:

```
<script type="text/javascript">
    var Amigos = new Array (6);
    Amigos[0] = "EVA";
    Amigos[1] = "ANDRES";
    Amigos[2] = "MIGUEL";
    Amigos[3] = "JAVIER";
    Amigos[4] = "JOSE";
    Amigos[5] = "CARMEN";
    for (i = 0; i < 6; i = i + 1)
    {
        document.write(Amigos[i] + " * ");
    }
</script>
```

El resultado de la ejecución de este programa será:

EVA*ANDRÉS*MIGUEL*JAVIER*JOSÉ*CARMEN*

1.8.7.4. Introducción de los elementos del Array desde la consola del usuario

Hasta el momento hemos trabajado con un *Array* cuyos datos se introducen directamente desde el código *JavaScript*. Pero en ocasiones necesitaremos que sea el usuario quien introduzca los elementos del *Array*.

A continuación estudiaremos un programa en el que el usuario introduce los datos que se guardan en el *Array* de manera secuencial, es decir, cada elemento a continuación del anterior.

```
<script type="text/javascript">
    // declaramos el array
    var Amigos = new Array (6);
    // solicitamos al usuario que inserte los datos
    for ( i = 0; i < 6; i++)
    {
        Amigos[i] = prompt("Introduce nombre del amigo "+i, "");
    }
    // presentamos en pantalla los datos introducidos
    for ( i = 0; i < 6; i++)
    {
        document.write(Amigos[i] + "*");
    }
</script>
```

Con *JavaScript* también podemos crear un *Array* e introducir sus elementos simultáneamente desde el código, declarando el nombre y enumerando a continuación en una lista los elementos:

```
var Amigos = new Array ("EVA", "ANDRES", "MIGUEL", "JAVIER",
"JOSE", "CARMEN");
```

El *Array* se dimensionará implícitamente en función del número de elementos que lo compongan. Por su parte, los elementos definidos así, se asociarán con el índice según la posición que ocupan en la lista, es decir, el primero será el 0, el segundo el 1, y así sucesivamente con todos.

1.8.7.5. Búsqueda en un Array

En los Arrays, podemos realizar búsquedas basándonos en los siguientes criterios:

- Búsqueda a partir del índice. Si conocemos cual es el índice, obtenemos el elemento).
- Búsqueda de un elemento para obtener su posición o simplemente para saber que existe.

La primera no plantea ningún problema ya que si conocemos el índice, el acceso al elemento correspondiente es automático, (por ejemplo, si buscamos el elemento 3 accedemos a él como `Amigos[3]`).

La segunda implica recorrer el Array, comparando uno a uno cada elemento con el valor que se busca hasta obtener dicho valor o llegar al final sin obtener un resultado satisfactorio. En este caso hay que emplear algunas líneas de programa:

```
...
var Vbusca = ... // Vbusca es la variable que suponemos
// contiene el valor a buscar
var Ultimo = NombreDelArray.length -1
i = 0;
while (NombreArray[i] != Vbusca && i < Ultimo)
{
    i = i + 1;
}
```

La salida del bucle se producirá por una de las siguientes circunstancias: o se ha llegado al último elemento; o bien, ha encontrado en valor buscado. Deberemos, por tanto, comprobar si realmente ha encontrado lo que buscaba o no. Para ello debemos hacer lo siguiente:

```
if (NombreArray[i] == Vbusca)
{
    // sentencias en caso de encontrar el elemento.
}
else
{
    // Tratamiento en caso de NO encontrar el elemento.
}
```

Vamos a verlo mejor con el siguiente ejemplo práctico. Vamos a pedir un nombre de amigo por teclado, si el nombre existe en la lista se visualizará el mensaje “Existe”, si no existe se visualizará el mensaje de “No existe”:

```
<script type="text/javascript">
// declaramos el Array
```

```

var Amigos = new Array ("EVA", "ANDRES", "MIGUEL",
"JAVIER", "JOSE", "CARMEN");

Ultimo = Amigos.length - 1;
Vbusca = "";
i = 0;
***** Entrada del nombre a buscar ****/
Vbusca = prompt("Introduce el nombre a buscar", " ");

***** Búsqueda ****/
i = 0;
while ( Amigos[i] != Vbusca && i < Ultimo)
{
    i = i + 1;
}
***** Comprobación y visualización ****/
if (Amigos[i] == Vbusca)
{
    alert("Existe: " + Vbusca +
" con el número: " + i);
}
else
{
    alert("No existe: " + Vbusca);
}

</script>

```

1.8.7.6. El objeto Array

El objeto *Array* es un objeto predefinido por el lenguaje que nos va a permitir construir *Arrays*, como ya hemos visto. Como todo objeto, dispone de una serie de propiedades y métodos.

Dentro de las propiedades hemos utilizado una que es la propiedad `length`, que devuelve el número de elementos del *Array*. De los métodos podemos destacar algunos como:

- ***join(separador)***. No genera una cadena con todas las cadenas de cada uno de los elementos del *Array* separadas por el separador especificado entre los parentesis.
- ***reverse()***. Invierte el orden de los elementos del *Array* devolviendo una cadena con los elementos del *Array* ordenados inversamente.
- ***sort()***. Ordena los elementos del *Array* siguiendo un orden alfabético.

Veamos un ejemplo que nos va a mostrar el uso de los métodos descritos anteriormente:

```
<script type="text/javascript">  
// declaramos el Array  
  
var Amigos = new Array ("EVA", "ANDRES", "MIGUEL",  
"JAVIER", "JOSE", "CARMEN");  
  
document.write(Amigos.join("*") +"  
");  
document.write(Amigos.reverse() +"  
");  
document.write(Amigos.sort() +"  
");  
  
</script>
```

Si ejecutamos el script, nos dará la siguiente respuesta:

EVA*ANDRÉS*MIGUEL*JAVIER*JOSÉ*CARMEN
CARMEN,JOSÉ,JAVIER,MIGUEL,ANDRÉS,EVA
ANDRÉS,CARMEN,EVA,JAVIER,JOSÉ,MIGUEL

1.9. Funciones

En muchas ocasiones nos encontraremos ante un script de una cierta dimensión o complejidad, el cual lo podemos dividir en partes más pequeñas, de forma que, cada una sirve para un determinado propósito, creándose de esta manera las funciones.

Las funciones nos van a permitir ejecutar una serie de instrucciones cuando se produce un evento (por ejemplo, cuando se pulsa un botón). Se pueden definir tantas funciones como sea preciso.

D **DEFINICIÓN**

Evento
Un evento es una acción del usuario ante la cual puede realizarse algún proceso (por ejemplo, el cambio del valor de un formulario o la pulsación de un enlace).

También nos permitirán que se ejecute el mismo código sin necesidad de insertar la secuencia de instrucciones repetidas veces.

D **DEFINICIÓN**

Funciones
Una función es un conjunto de instrucciones a las que se asigna un nombre, devuelve o no un valor y se puede ejecutar tantas veces como se desea con solo llamarla con el nombre asignado.

A la hora de trabajar con funciones debemos distinguir dos aspectos: la creación de la función o definición de la misma y la utilización de la función o llamada a esta.

1.9.1. Creación de una función

Para crear una función debemos determinar los siguientes aspectos:

- El nombre de la función.
- Los parámetros o valores sobre los que actuará, es decir, los argumentos.
- Las acciones que deberá realizar.
- El valor que devolverá, aunque puede no devolver ningún valor.

Y aplicar el siguiente formato:

```
function nombreFuncion (listaArgumentos)
{
    instrucciones...
    ...
    [return valorRetorno;]

}
```

Dónde:

- `nombreFuncion` será el identificador válido que nos servirá para realizar las llamadas a la función.
- `listaArgumentos` será la lista de variables (separadas por comas) que recogerán los valores que se pasen en la llamada a la función.
- `valorRetorno` será la expresión cuyo valor devolverá la función.

Vamos a ver un pequeño ejemplo de una función que la llamamos `suma`, la cual recibe dos valores (`A, B`) y devuelve su suma:

```
function suma (A, B)
{
    var C;
    C = A + B;
    return C;
}
```

Debemos tener en cuenta, que cuando el navegador encuentra la definición de una función, la va a cargar en la memoria, pero no la va a ejecutar hasta que se produzca una llamada a dicha función.

1.9.2. ¿Dónde declarar una función?

Las funciones pueden declararse en cualquier parte de una página HTML entre las etiquetas `<script>` y `</script>` pero siempre debemos tener en cuenta que no pueden definirse:

- Dentro de otra función.
- Dentro de una estructura de control.

Normalmente se definen dentro de la cabecera de una página HTML, así cuando se carga la página en el navegador del cliente estarán disponibles para ser utilizadas.

Vamos a ver un ejemplo de función con la que visualizaremos la hora en formato Hora:Minutos.

```
<script type="text/javascript">
    function Hora()
    {
        var hoy =new Date()
        document.write(hoy.getHours() , ":", hoy.getMinutes())
    }
</script>
```

Podemos observar en el anterior ejemplo, que la función se llama `Hora()`, no recibe argumentos y no devuelve ningún valor. Define una variable `hoy` que utiliza el objeto `Date()` para poder obtener la hora `hoy.getHours()` y los minutos `hoy.getMinutes()`. Este objeto lo trataremos en el capítulo de objetos.

1.9.3. Llamada a una función

Ya sabemos cómo y dónde declarar una función. Para realizar las llamadas que producirán la ejecución de la misma como si se tratase de cualquier instrucción, deberemos hacerlas siguiendo el formato:

```
nombreFuncion(listaParametros);
```

Debemos tener en cuenta que la llamada a una función nos puede devolver un valor y que debemos hacer algo con él, ya sea escribirlo, asignarlo a una variable, etcétera.

Por ejemplo, para hacer una llamada a la función `suma` y escribir el valor devuelto en el documento actual escribiríamos lo siguiente:

```
document.write(suma(2, 3));
```

Por ejemplo, para llamar a la función `Hora` sólo tendremos que escribir su nombre y los paréntesis vacíos porque no lleva argumentos `Hora()`

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Funciones</title>
<script type="text/javascript">
    function Hora()
    {
        var hoy =new Date();
        document.write(hoy.getHours() , ":", hoy.getMinutes());
    }
</script></head>
```

```
<body>
<script type="text/javascript">
    document.write("Hora actual: ");
    Hora();
</script>
</body>
</html>
```

1.9.4. Argumentos



Argumentos

Son variables locales a la función que toman un valor determinado cuando se les invoca.

Las funciones con argumentos generan resultados distintos dependiendo de los valores que tomen los argumentos cuando se llama a la función.

La función `suma (A, B)` expuesta a continuación recibe dos valores y devuelve su suma:

```
<head>
<title>Funciones</title>
<script type="text/javascript">
    function suma (A, B) {
        var C;
        C = A + B;
        return C;
    }
</script>
</head>
<body>
<script type="text/javascript">
    document.write("Llamada a la función Suma con argumentos: ");
    document.write(suma(2, 3));
</script>
</body>
</html>
```

Cuando lo ejecutemos obtendremos el siguiente resultado:

Llamada a la función Suma con argumentos: 5

1.9.5. Consideraciones con el uso de funciones

Cuando usamos funciones en *JavaScript*, debemos tener en cuenta los siguientes aspectos:

- Que tanto las funciones como las llamadas deben ir entre etiquetas `<script>` `</script>`.
- Que la cláusula `return` produce la salida de la función devolviendo (opcionalmente) un valor.
- Debemos recordar que *JavaScript* permite escribir varias cláusulas `return` en una misma función.

```
function mayor(A,B)
{
    if (A >= B) {
        return A;
    }
    else {
        return B;
    }
}
```

- Debemos tener en cuenta que se pueden escribir funciones que no devuelvan ningún valor, sino que solamente realizan una o varias acciones. También se pueden escribir funciones que no tengan parámetros.

Si usamos funciones con parámetros, y le pasamos un número menor de parámetros, estos que no han obtenido valor como resultado de la llamada quedarán con el valor `null`. Por ejemplo, si la llamada a la función suma se realiza con un solo número el resultado será:

Llamada a la función Suma con argumentos: NaN

Para solucionar esto, podemos comprobar el número de parámetros que se ha pasado a una función mediante `arguments.length` tal como podemos apreciar en el siguiente ejemplo:

```
<script type="text/javascript">
    function suma(A,B)
    {
        var C;
        if (arguments.length < 2) {
            B = 0;
```

```

        }
        if (arguments.length < 1) {
            A = 0;
        }
        C = A + B;
        return C;
    }

</script>
```

1.9.6. Variables locales y globales

Debemos saber diferenciar que son las variables locales y globales. Son variables locales aquellas que se declaran en la definición de la función, y sólo son accesibles en el ámbito de la función donde se declaran.

Las variables globales se declaran fuera de la declaración de cualquier función y se puede tener acceso a ellas desde cualquier parte del documento.

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Funciones</title>
<script type="text/javascript">
    var C; //variable global,
    C=0;
    function suma (A,B) {
        C = A + B;
        return C;
    }
    function resta (A,B) {
        return C+A-B;
    }
</script>
</head>
<body >
<script type="text/javascript">
    document.write("Sumamos dos valores: ");
    document.write(suma(2,3));
    document.write('<BR>');
    document.write("Restamos dos valores: ");
    document.write(resta(10,20));
</script>
</body>
```

```
</html>
```

Si ejecutamos el script, obtenemos la siguiente salida:

```
Suma con argumentos: 5
Restamos dos valores -5
```

1.9.7. Funciones predefinidas

Existen algunas funciones propias del lenguaje que nos van a permitir convertir cadenas a enteros o reales, evaluar una expresión, etc.; aunque algunas de ellas ya las hemos visto en puntos anteriores, volveremos a definirlas y explicarlas en este apartado:

- ***parseFloat(cadena)***: esta función nos devuelve un número en coma flotante a partir de la cadena especificada. La función finalizará la conversión cuando encuentre un carácter que no sea un dígito (0-9), un punto decimal (.) o una “e”. Ejemplos:

```
parseFloat("27.11.2011"); //devuelve 27.11
parseFloat("20H30M07S"); //devuelve 20
parseFloat("345.6e5"); //devuelve 34560000
parseFloat("VUELO506"); //devuelve NaN (Not a Number)
```

- ***parseInt(cadena)***: nos devolverá un número entero a partir de la cadena especificada. La función finalizará la conversión cuando encuentre un carácter que no sea un dígito (0-9). Ejemplos:

```
parseInt("27.11.2000"); //devuelve 27
parseInt("20H30M07S"); //devuelve 20
parseInt("345.6e5"); //devuelve 345
parseInt("VUELO506"); //devuelve NaN (Not a Number)
```

- ***parseInt(cadena, base)***: si no se indica nada al respecto, se considerará como base la decimal (excepto cuando el número comienza con 0x o con 0); pero en este formato se puede añadir un segundo argumento que indicaría la base en la que creemos que está el número que queremos convertir. Ejemplos:

```
parseInt("27", 8); //devuelve 23
parseInt("11111111", 2); //devuelve 255
parseInt("A2", 16); //devuelve 162
```

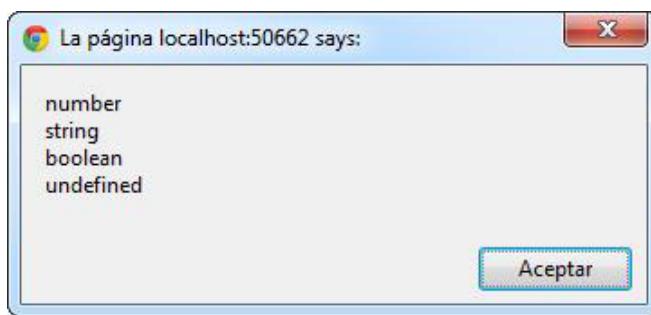
- ***toString(argumento)***: convierte a cadena el dato especificado en argumento. Normalmente se utiliza para convertir números a cadena pero también funciona con otros tipos como Booleano, etcétera.

Su utilización es restringida ya que *JavaScript* aplica una conversión automática de tipos a cadena cuando encuentra expresiones de concatenación en las que hay diversos tipos de datos.

- **typeof(argumento):** nos devolverá una cadena que indica el tipo del argumento (number, string, boolean, undefined). El siguiente programa muestra la utilización de esta función:

```
<script type="text/javascript">
    V1 = 10;
    V2 = 'a';
    V3 = true;
    var V10;
    alert(typeof(V1) + "\n" + typeof(V2) + "\n" +
        typeof(V3) + "\n" + typeof(V10));
</script>
```

Si ejecutamos el script nos dará lo siguiente:



- **escape(cadena):** devuelve la secuencia de códigos de escape de cada uno de los caracteres de la cadena que le pasemos como argumento. Estos códigos serán devueltos con el formato %NN, donde NN es el código de escape en hexadecimal del carácter.
- **unescape(cadena):** devuelve la cadena formada por los caracteres cuyo código le especificaremos. Los códigos deben ser especificados con el mismo formato con el que escape los proporciona: %NN, donde NN es el código en hexadecimal.
- **isNaN(argumento):** con esta función evaluaremos el argumento que se le pasa y nos devolverá false si es un número; en caso contrario, devuelve true. Por ejemplo:

```
isNaN("seas"); devuelve true.  
isNaN("10"); devuelve false.
```

- **eval(expresión):** devuelve el resultado de evaluar la expresión pasada como parámetro. Por ejemplo `eval("2+3")`; nos devuelve 5.

1.9.8. Funciones de cadena de texto

Cuando hablamos de las cadenas de texto, debemos saber que JavaScript nos provee a través del objeto *String* un conjunto de funciones que van a permitir al programador multitud de operaciones con las cadenas de texto.

Aunque ya se han visto algunas de ellas durante este temario, vamos a verlas todas juntas

charAt

La función *charAt* la vamos a usar para conocer cuál es el carácter que se encuentra en la posición índice de la cadena.

```
caracter = cadena.charAt(índice);
```

Los índices, al igual que en las matrices, comienzan en 0.

```
caracter = "seas".charAt(3);
```

Esta sentencia nos devolverá en la variable *caracter* el valor “a”, que es el que ocupa la posición 3.

Length

Utilizaremos la función *length* para conocer la longitud de una cadena.

```
longitud = "seas".length;
```

La salida de esta sentencia es que la variable *longitud* tiene un valor de 4.

IndexOf

Utilizaremos la función *indexOf* para saber si una cadena está contenida en otra, y si es así, nos devuelve la posición en la cadena de búsqueda donde se inicia la contención.

```
posicion = "estudiosabiertos".indexOf("os");
```

La salida de esta sentencia es que la variable *posicion* tiene un valor de 7.

Si no se encuentra la subcadena dentro de la cadena donde se busca, nos devolverá el valor -1.

Además, esta función tiene un segundo parámetro, que es opcional, y que sirve para indicar la posición a partir de la cual se ha de buscar la subcadena.

```
posicion = "estudiosabiertos".indexOf("os", 3);
```

En este ejemplo se comenzaría a buscar desde la posición 3, es decir, desde la “t”.

Substr

Usaremos la función `substr` para obtener un subconjunto de la cadena de texto.

```
cadena = "Texto".substr(inicio, longitud);
```

Un ejemplo sería:

```
cadena = "estudiosabiertos".substr(3,5);
```

Empezando desde el carácter de la posición 3 y con una longitud de 5 caracteres, la salida de esta sentencia es que `cadena` tendrá un valor de “`udios`”.

Slice

La función `slice` es muy similar a la función `substr` descrita en el punto anterior, pero la diferencia es que el segundo argumento en lugar de ser la longitud de la subcadena a extraer es la posición del último carácter a extraer, sin incluirlo. Así, el ejemplo anterior sería como sigue:

```
cadena = "estudiosabiertos".slice(3,5);
```

En este caso es el mismo rango de números. El carácter que ocupa la posición 5 es “`i`”, y como no se incluye ese carácter, la cadena resultante sería “`udi`”.

El segundo parámetro es opcional, y si no se pone, se llega hasta el final de la cadena.

Substring

Es otra función más para obtener una subcadena a partir de otra. Funciona exactamente igual a `slice` pero el segundo parámetro sí que es obligatorio. Esta función, junto con `substr`, es de las más usadas.

```
cadena = "estudiosabiertos".substring(3,5);
```

toLowerCase

Esta función se utiliza para convertir a minúsculas la cadena a la que aplica:

```
minusculas = "EstudiosAbiertos".toLowerCase();
```

El resultado será que `minusculas` será igual a “`estudiosabiertos`”.

toUpperCase

Función que se utiliza para convertir a mayúsculas la cadena a la que aplica.

```
mayusculas = "EstudiosAbiertos".toUpperCase();
```

El resultado será que `mayusculas` será igual a “`ESTUDIOSABIERTOS`”.

Estas dos funciones son de uso frecuente en las comparaciones entre cadenas, ya que normalmente no nos interesarán las diferencias entre mayúsculas o minúsculas.

1.10. Objetos

La Programación Orientada a Objetos, o POO, trata de vincular el mundo real con el mundo del software. Se intenta modelizar el mundo real de una forma cercana al propio mundo real.

En lo que llevamos visto de JavaScript ya se han mostrado varios objetos.

Por ejemplo, cuando hablamos de *Date*, estamos hablando de un objeto; cuando hablamos de las cadenas de texto, de los *String*, estamos hablando de un objeto y cuando, por ejemplo, hemos hablado de funciones como *substr*, estas funciones son en realidad métodos del objeto *String*.

Como se puede ver, hemos ido incluyendo los objetos en nuestros pequeños códigos y lo hemos hecho de una forma natural ya que la POO, lejos de ser compleja, es una modelización natural del mundo.

La orientación a objetos se empezó a desarrollar paralelamente con el desarrollo de los entornos gráficos de usuario (GUI), así por ejemplo cuando estamos ante una herramienta de dibujo y hacemos clic en un ícono para pintar un cuadrado hará que el puntero del ratón pueda dibujar un cuadrado en la ventana del dibujo.

La orientación de objetos se centra en identificar los objetos procedentes del dominio de la aplicación que se va a estudiar y en especificar lo que es y lo que hace más es especificar la forma en que se utiliza. Un objeto es un concepto o una vista abstracta de una entidad, es como una variable en algunos aspectos, su objetivo es representar una cosa, por ejemplo un objeto podría ser un botón o un campo de entrada de un documento HTML.

Cualquier objeto posee unas características que lo define, estas características se llaman *propiedades*, cada propiedad tiene un valor de algún tipo unido a él. Con los objetos podemos hacer cosas, necesitamos saber los nombres de las cosas que podemos hacer y los parámetros que necesitamos para hacerlas. Cada una de estas cosas se llama *método*.

Por ejemplo en la sentencia:

```
document.write("Hola");
```

Tenemos el objeto `document` y el método `write` que requiere un parámetro. JavaScript utiliza el punto para realizar instrucciones utilizando los objetos. En esta otra línea:

```
document.FORMULARIO.RESUL.value;
```

Tenemos la propiedad `value`, de un objeto `RESUL`, de un objeto `FORMULARIO` de un objeto `document`.

1.10.1. Jerarquía de los objetos

Podemos decir que *JavaScript* no es un lenguaje orientado a objetos pero sí es un lenguaje basado en objetos. En *JavaScript* podemos considerar a los objetos como colecciones de propiedades y métodos. Admite tres tipos de objetos:

- **Los objetos del navegador:** window, frame, location, history y navigator.
- **Los objetos predefinidos:** Array, Boolean, Date, Function, Math, Number, Object y String.
- **Los objetos creados por el usuario:** para dar solución a sus necesidades.

Referencia a los objetos

En *JavaScript* los objetos y las matrices se manipulan igual. Podemos referenciar a los miembros de un objeto (propiedades y métodos) de la siguiente forma:

- Por su nombre o notación de punto:

NombreObjeto.NombrePropiedad

NombreObjeto.NombreMétodo(argumentos)

- Por su posición que ocupa en la jerarquía de objetos o notación de array:

NombreObjeto[posición]

NombreObjeto[NombrePropiedad]

- Usando la palabra clave `this` que hace referencia al objeto actual. Si se utiliza fuera del contexto de cualquier objeto hace referencia al objeto `window`. Admite referencia por nombre o por posición:

`this.NombrePropiedad`

`this.NombreMétodo(argumentos)`

`this[posición]`

`this[NombrePropiedad]`

Vamos a ver un ejemplo que nos mostrará el título de la página Web y la URL actual de diversas formas. Para obtener el título utilizamos el objeto `document` y la propiedad `title`. Para obtener la URL utilizamos el objeto `window` y la propiedad `location`. Estos objetos se verán a lo largo del tema.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Objetos</title>
```

```
<script type="text/javascript">
    function ver(dato) {
        document.write(dato);
    }
</script>
</head>
<body>
<script type="text/javascript">
ver("TITULO DEL DOCUMENTO: <BR>");
ver( document.title); ver(" ==>document.title<BR>");
ver(document["title"]); ver(" ==>document[\"title\"]<BR>");
ver(this.document.title); ver(" ==>this.document.title<BR>");
ver("URL del DOCUMENTO: <BR>");
ver(window.location);
</script>
</body>
</html>
```

La función `ver(dato)` visualiza el contenido del parámetro que recibe. La salida generada es la siguiente:

```
TITULO DEL DOCUMENTO:
Objetos ==>document.title
Objetos ==>document["title"]
Objetos ==>this.document.title
URL del DOCUMENTO:
http://localhost:50662/Website2/HTMLPage.htm
```

Los objetos del navegador

La jerarquía muestra el camino para poder referenciar a un objeto con respecto a todos los objetos contenidos en la ventana del navegador.

El siguiente gráfico muestra la jerarquía de los objetos del navegador:

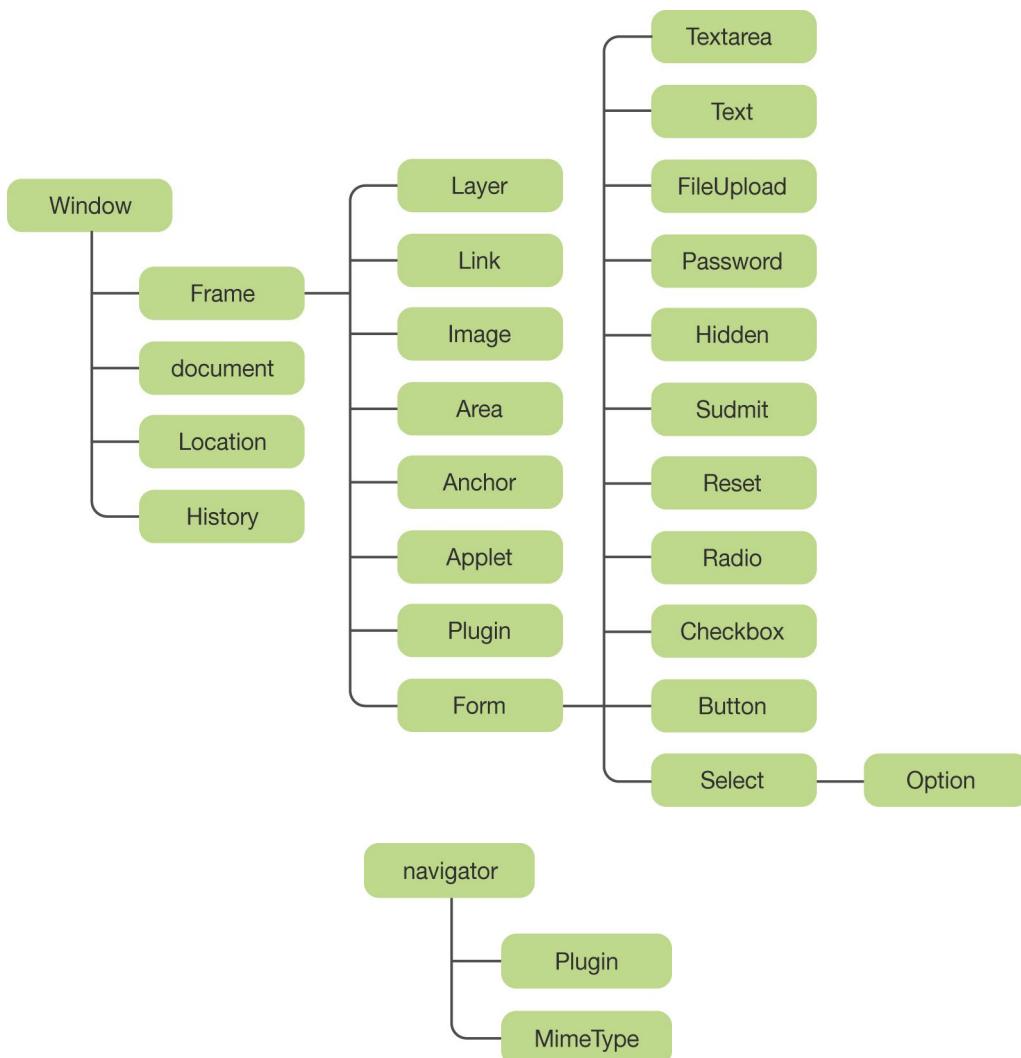


Figura 1.13. Jerarquía de objetos del navegador.

Se pueden observar dos jerarquías de objetos, una la del objeto *window* formada por todos los elementos presentes en la visualización de un documento HTML, y otra la del objeto *navigator* relacionado con las características del navegador.

La mayoría de los objetos del navegador tienen una correspondencia con las etiquetas HTML, así la etiqueta <FRAME> se corresponde con el objeto *Frame*, la etiqueta <FORM> con el objeto *Form*, la etiqueta con el objeto *Link*, la etiqueta <INPUT TYPE=BUTTON> con el objeto *Button*, la etiqueta <BODY> con el objeto *Document*, etc. Sólo los objetos *window*, *history* y *location* no tienen una correspondencia directa con las etiquetas HTML.

Según esta jerarquía, podemos decir por ejemplo que el objeto caja de texto *text* es un objeto que está dentro del objeto *form*, que a su vez está dentro del objeto *document* y que este está dentro del objeto *window*. Para hacer referencia a la caja de texto, tendremos que escribir: `window.document.form.text`.

En la mayoría de los casos podemos ignorar la referencia a la ventana actual (*window*), pero será necesaria cuando estemos utilizando múltiples ventanas o cuando usemos *frames*.

El uso de *arrays* como elementos dentro de la estructura jerárquica dentro del modelo de objetos de *JavaScript* es muy importante, puesto que proporcionan una manera alternativa para referenciar a los elementos.

Así, podemos referenciar a un objeto por su posición dentro del *array* que lo contiene. Los siguientes *arrays* son los más utilizados para referenciar los componentes de un documento:

- ***Frames[posición]***: contiene los diferentes marcos del documento.
- ***Links[posición]***: contiene los diferentes enlaces externos de un documento.
- ***Images[posición]***: contiene las diferentes imágenes de un documento.
- ***Forms[posición]***: contiene los diferentes formularios de un documento.
- ***Elements[posición]***: contiene los diferentes elementos de un formulario.
- ***options[posición]***: contiene las diferentes opciones de un objeto select.

En el siguiente ejemplo vamos a mostrar los nombres de algunos de los objetos del documento HTML utilizando la notación de *array*. El código es el siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>OBJETOS - EJEMPLO 2</title>
<script type="text/javascript">

    function ver(dato) {
        document.write(dato);
    }

    function objetos() {
        ver("<BR><CENTER>");
        ver("nombre de la primera imagen:");
        ver("+document.images[0].name+"<BR>");
        ver("nombre de la segunda imagen:");
        ver("+document.images[1].name+"<BR>");
        ver("numero de imagenes:");
        ver("+document.images.length+"<BR>");
        ver("nombre del segundo vinculo:");
        ver("+document.links[1].name+"<BR>");
        ver("numero de elementos del formulario:");

    }
}</script>
```

```

"+document.forms[0].elements.length+"<BR>");

}

function vertexto() {
    ver("contenido del texto:
"+document.forms[0][0].value+"<BR>");

}

</script>

</head>

<body>

<p align="center">

Seas1 

<a href="http://www.seas.es" name="linkSeas1">--SEAS--</a>

Seas3

</p><br />

<p align="center">

<a href="http://www.seas.es" name="linkSeas2">SEAS</a>&ampnbsp&ampnbsp

Seas2 

<center>

<table border="1" cellpadding="0" cellspacing="0" bgcolor="#FFFF89">

<tr>

<td>

<p align="center"><b>FORMULARIO </b><p>

<form name="FORMULARIO">TEXTO

<input type="text" name="TEXTO" size="20" />

<input type="submit" value="Ver texto" name="BOTON"
onclick="vertexto()"/>

</form>

</td>

</tr>

</table>

</center>

</p>

```

```

<script type="text/javascript">
    objetos();
</script>
</body>
</html>

```

La función `objetos()` se encarga de visualizar los nombres de los objetos. La función `ver(dato)` se encargará de visualizar el dato en pantalla. Al pulsar en el botón del formulario se visualiza el contenido del texto escrito en la caja de texto, esta misión la realiza la función `verTexto()`.



Para obtener el nombre de un objeto se utiliza la palabra `name`. Por ejemplo con la sentencia `document.images[0].name` obtenemos el nombre de la primera imagen del documento HTML.

Veamos la ejecución del código:

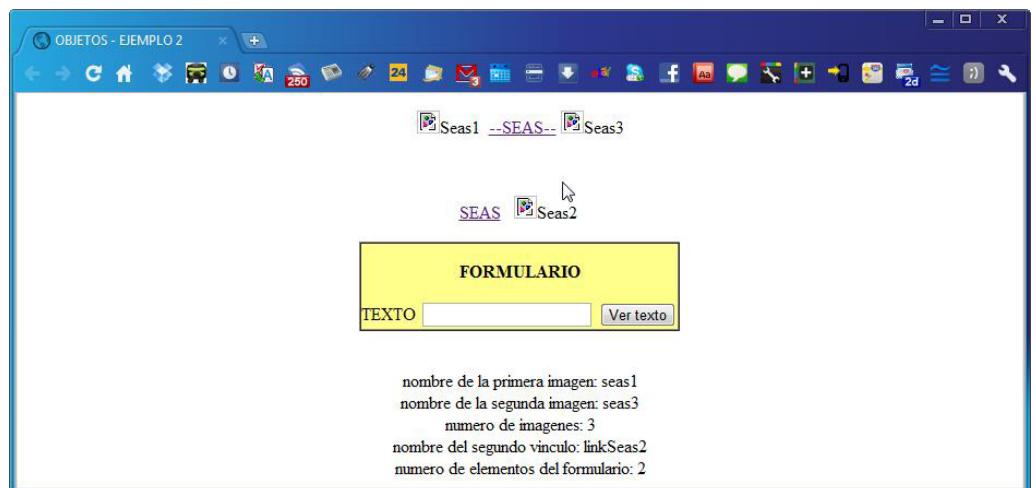


Figura 1.14. Ejecución del código ejemplo2.

Veamos a continuación una breve descripción de los objetos del navegador:

- **Objeto window:** representa la ventana del navegador o cualquiera de sus marcos, es el de mayor jerarquía. Por cada etiqueta BODY o FRAMESET se genera un objeto `window`.
- **Objeto document:** es el que tiene el contenido de toda la página que se está visualizando. Esto incluye el texto, imágenes, enlaces, formularios... y gracias a este objeto vamos a poder añadir dinámicamente contenido a la página o hacer cambios, según nos convenga.

- **Objeto form:** contiene los elementos necesarios para generar formularios. Los formularios se agrupan en un *array* dentro de *document*. Cada elemento de este *array* es un objeto de tipo *form*.
- **Objeto frame:** un objeto *frame* se comporta igual que un objeto *window* y contiene sus mismas propiedades y métodos.
- **Objeto history:** contiene la información de las URL que el usuario ha visitado desde esa ventana.
- **Objeto location:** contiene información sobre la URL actual.

Objetos predefinidos

Estos objetos están relacionados con el propio lenguaje y nos permitirán manejar nuevas estructuras de datos y utilidades. Algunos ya los hemos utilizado en unidades anteriores como el objeto Array, Date y Function entre otras:

- **Objeto Array:** nos permite gestionar matrices, vectores o arrays.
- **Objeto Boolean:** nos permite crear objetos lógicos que representan el valor verdadero (true) o falso (false).
- **Objeto Number:** nos permite trabajar con datos numéricos.
- **Objeto String:** nos permite trabajar con cadenas de caracteres.
- **Objeto Function:** ya hemos utilizado este objeto para crear nuevas funciones.
- **Objeto Math:** permite trabajar con las funciones matemáticas más usadas.
- **Objeto Object:** proporciona funcionalidades comunes a todos los objetos JavaScript.
- **Objeto Date:** contiene los objetos necesarios para trabajar con los valores de fecha y hora. En capítulos anteriores ya hemos visto como se usaban.

En el siguiente ejemplo vamos a ver cómo funcionan los objetos *Array*, *String* y *Math* con algunos de sus métodos: *cad* es un objeto *String*, *nombres* un objeto *Array* y a *numero* le aplicamos algunos métodos del objeto *Math*:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>OBJETOS - EJEMPLO 2</title>
<script type="text/javascript">

function objetosPredefinidos () {
    var cad= "Objetos predefinidos";
    var nombres =new Array("Juan", "Ana", "Julio", "Maria", "Alicia");
    var numero = 36;
```

```

        document.write("Longitud de la cadena: "+cad.length+"<BR>");

        document.write("Mayúscula: "+cad.toUpperCase()+"<BR>");

        document.write("De color rojo: "+cad.fontcolor("#FF0000")+"<P>");

        document.write("Antes de ordenar el array: " + nombres.
join(',')+"<BR>");

        document.write("Nombres ordenados: "+nombres.sort()+"<BR>");

        document.write("Ordenados en orden inverso: " + nombres.
sort().reverse()+"<P>");

        document.write("Raiz cuadrada de 36: " +Math.sqrt(numero)+"<BR>");

        document.write("36 elevedo a 5: " + Math.pow(numero,5)+"<BR>");

    }

</script>

</head>

<body>

<script type="text/javascript">

    objetosPredefinidos();

</script>

</body>

</html>

```

Si abrimos la pagina, ejecuta la siguiente salida:

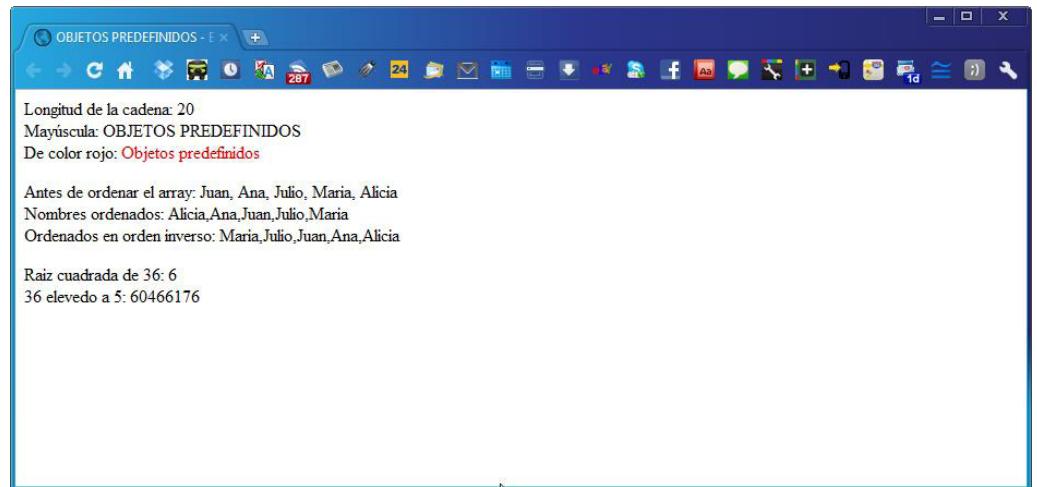


Figura 1.15. Salida por pantalla de ejemplo de objetos predefinidos.

Objetos personalizados

JavaScript nos permite crear nuestros propios objetos con propiedades y con métodos. Recordemos que un objeto es una entidad que posee unas ciertas características, llamadas propiedades, y que tiene asociadas determinadas operaciones, llamadas métodos.

Así podemos definir el objeto *Alumno* con varios atributos: nombre, apellidos, curso y nota; y varios métodos como pueden ser: obtener la nota del alumno, obtener todos los datos del alumno, etc. A la hora de crear un objeto personalizado necesitamos una función que define cómo es el objeto y cómo se comporta. Así, definimos el objeto *Alumno* de la siguiente manera:

```
function Alumno (Nombre, Apellidos, Curso, Nota) {  
    this.nombre=Nombre;  
    this.apellidos=Apellidos;  
    this.curso=Curso;  
    this.nota=Nota;  
    this.Obtener_nota=Obtener_nota;  
    this.Obtener_datos=Obtener_datos;  
}
```

Dónde:

- El nombre de la función (*Alumno*) se utilizará como nombre de objeto.
- Los parámetros de la función representan los valores que se asignarán a las propiedades del objeto (*Nombre*, *Apellidos*, *Curso*, *Nota*).
- Se utiliza la palabra reservada *this* seguida de un punto para asociar propiedades al objeto (*this.propiedad*).
- También se utiliza la palabra reservada *this* para añadir métodos al objeto (*this.Obtener_nota*).
- A los atributos se les asigna valores (parámetros que se pasan a la función) y los métodos se crean fuera de la declaración del objeto.

Se han definido dos métodos: *Obtener_nota* y *Obtener_datos*, que se definen fuera de la definición del objeto:

```

function Obtener_nota(){ //devuelve la nota
    return (this.nota);
}

function Obtener_datos(){ //devuelve datos del alumno
    var datos= "Nombre : " + this.nombre +
    "  
Apellidos: " + this.apellidos +
    "<br>Curso: " + this.curso +
    "<br>Nota: " + this.nota;
    return (datos);
}

```

Para utilizar este objeto creamos una instancia de la siguiente forma:

```
Alicia=new Alumno("Alicia","Perez Perez",1,7);
```

Para hacer referencia a los métodos y atributos:

```

Alicia.curso=2;
document.write(Alicia.Obtener_datos());
document.write("<br>La nota de " + Alicia.nombre +" es:
"+ Alicia.Obtener_nota());

```

Hemos cambiado el valor de la propiedad `curso` del objeto `Alicia` y los mostramos en la ventana. Si ejecutamos veremos la siguiente salida:

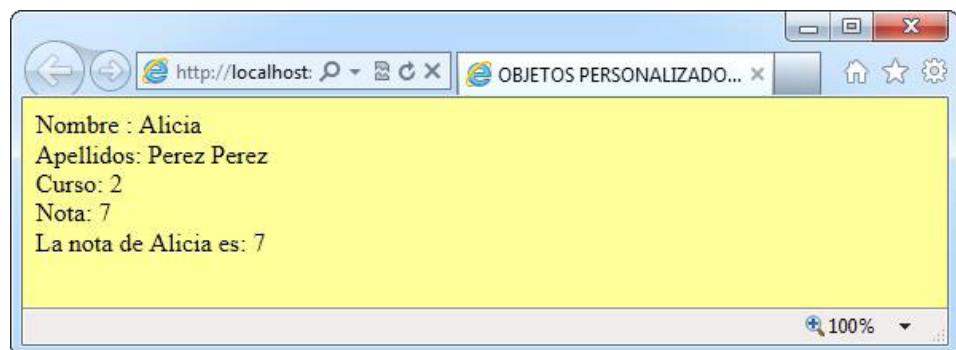


Figura 1.16. Vista del ejemplo1 de objetos personalizados.

1.10.2. Propiedades y métodos de los objetos del navegador

En este apartado veremos las propiedades y métodos de los objetos del navegador, empezaremos por el objeto `window` que es el de nivel más alto en la jerarquía. Se describirán las propiedades y métodos más usados.

Objeto Window

En el objeto ventana es donde van a acontecer todas las operaciones del documento. Las propiedades y métodos se resumen en la siguiente tabla, incluiremos también los manejadores de eventos aunque hablaremos de ellos más adelante.

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
closed	alert(mensaje),	
defaultStatus	open(), close(),	
frames	confirm(mensaje), prompt(), moveBy(x,y), moveTo(x,y),	
history	print(), setTimeout(),	
length	resizeBy(x,y), resizeTo(ancho,alto),	onLoad=
location, name,	scroll(x,y), scrollBy(x,y),	onUnload=
parent , opener	scrollTo(x,y)	
self, status	clearInterval() , setInterval()	
top	clearTimeout() , setTimeout()	

Figura 1.17. Propiedades y métodos del objeto Window.

Para crear una ventana nueva escribiremos:

```
nueaventana=window.open();
```

Para acceder a las propiedades o métodos lo haremos de la siguiente manera:

```
window.propiedad;  
window.método();  
self.propiedad;
```

Explicamos ahora las propiedades del objeto *Window*:

- **closed:** es un valor booleano que nos dice si la ventana está cerrada (closed = true) o no (closed = false).
- **defaultStatus:** contiene el texto por defecto que aparece en la barra de estado del navegador.
- **frames:** es un array que representa a los objetos frame contenidos en la ventana (frames[0], frames[1], ...). El orden de aparición (frame[0],...) es el orden en que se definen en el documento HTML.
- **history:** se trata de un array que representa las URLs visitadas por la ventana (están almacenadas en el historial).
- **length:** número de frames que contiene la ventana actual.
- **location:** cadena con la URL de la barra de dirección.

- ***name***: nombre de la ventana o del frame actual.
- ***opener***: es una referencia al objeto window que abrió la ventana si la ventana fue abierta usando el método open.
- ***parent***: referencia al objeto window que contiene el frameset (padre de esta ventana).
- ***self***: es un nombre alternativo de la ventana actual.
- ***status***: cadena con el mensaje que tiene la barra de estado.
- ***top***: nombre alternativo del objeto window padre de esta ventana.

Veamos un ejemplo con algunas de las propiedades del objeto window:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Propiedades Obj. window</title>
<script type="text/javascript">
    window.name="objeto window";
    window.status="Esto es un ejemplo";
    document.write("¿Está la Ventana cerrada? "+window.closed+"\n");
    document.write("Nombre de ventana: "+window.name+"\n");
    document.write("Mensaje de la barra de estado: "+window.
status+"\n");
    document.write("Valor de self: "+window.self.name+"\n");
    document.write("Valor de top: "+window.top.name+"\n");
    document.write("Valor de location: "+window.location+"\n");
</script>
</head>
<body>
<script type="text/javascript">
    objetosPredefinidos();
</script>
</body>
</html>
```

Si ejecutamos el script nos dará la siguiente salida:

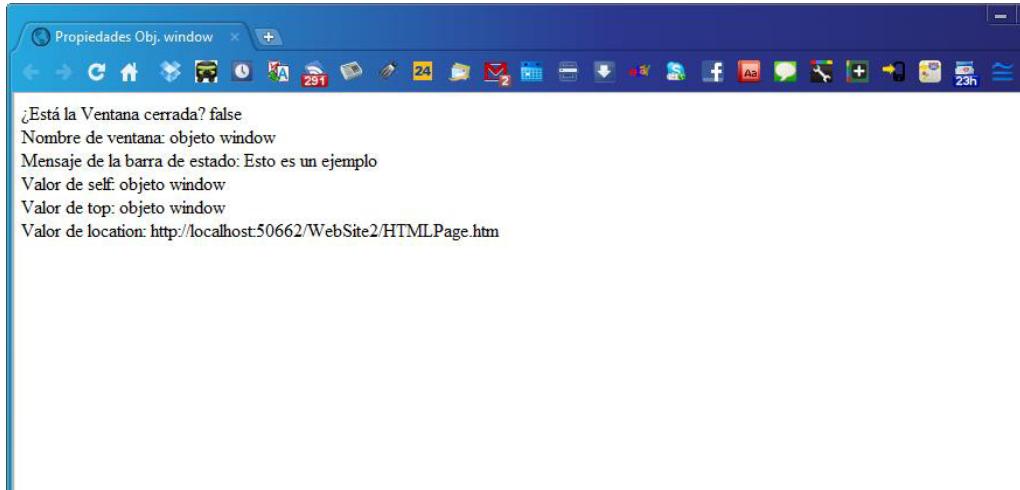


Figura 1.18. Salida ejemplo propiedades objeto Window.

Veamos otro ejemplo, el cual creará una ventana con un ancho y un alto que se piden por teclado. Al hacer clic en el botón “Crear Ventana” se creará la ventana con las especificaciones dadas y aparecerá un texto, al hacer clic en el botón “Cerrar Ventana” se cerrará la ventana actual y la ventana creada:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>OBJETO WINDOW - ventanas</title>
<script type="text/javascript">

    var ventana;

    function CrearVentana() {
        var opciones="left=100,top=100,";
        opciones=opciones +"width="+document.FORMULARIO[0].value+",";
        opciones=opciones +"height="+document.FORMULARIO[1].value;
        ventana = window.open("", "", opciones);
        ventana.document.write("Ventana nueva con el tamaño ");
    }

    function CerrarVentana() {
        if(ventana && !ventana.closed)
            ventana.close();
        window.close();
    }
}
```

```

</script>
</head>
<body>
<p align="center"><b> INTRODUCE EL ANCHO Y EL ALTO DE LA VENTANA:</b></p>
<center>
<table border="0" cellpadding="0" cellspacing="0">
<tr>
<td bgcolor="#FFFF96">
<form name="FORMULARIO">
<p><b>ancho de la ventana:</b><input type="text" name="ancho" size="10"/></p>
<p><b>alto de la ventana:</b><input type="text" name="alto" size="10"/></p>
<input type="button" value="Crear Ventana" name="B1" onclick="CrearVentana()">
<input type="button" value="Cerrar Ventana" name="B2" onclick="CerrarVentana()">
</form>
</td>
</tr>
</table>
</center>
</body>
</html>

```

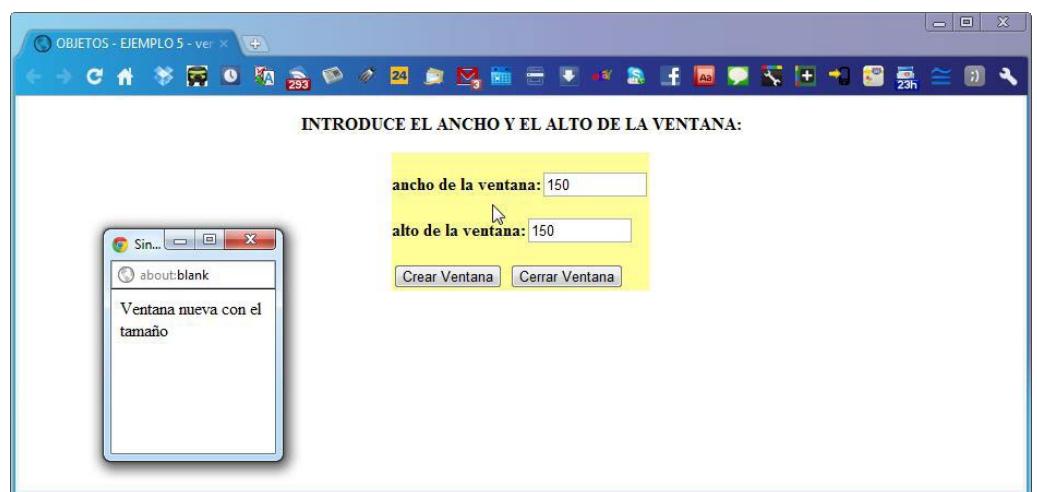


Figura 1.19. Salida ejemplo propiedades objeto Window.

Las función `CrearVentana()` crea la ventana. La función `CerrarVentana()` cierra las dos ventanas, la ventana actual y la ventana creada. Observa la sentencia siguiente: `if(ventana && !ventana.closed)` comprueba si la ventana creada existe y además que no esté cerrada, si se cumplen las condiciones se cierra dicha ventana.



NOTA

Para obtener el valor de un objeto utilizamos la palabra `value`. Por ejemplo con la sentencia:

```
document.FORMULARIO[0].value
```

Obtenemos el valor introducido en el primer elemento del formulario cuyo nombre es FORMULARIO.

A continuación veamos algunos `métodos` para mover y redimensionar ventanas, son los siguientes:

- **`resizeBy(x,y)`:** aumenta/disminuye el tamaño actual de la ventana en el número de píxeles indicados en x e y.
- **`resizeTo(ancho,alto)`:** ajusta el tamaño de la ventana a los valores indicados por la ventana.
- **`moveBy(x,y)`:** mueve la ventana actual el número de píxeles especificados por (x,y). A partir de Netscape 4 (JavaScript 1.2).
- **`moveTo(x,y)`:** mueve la ventana actual a las coordenadas (x,y) a partir de JavaScript 1.2.
- **`scroll(x,y)`:** hace que en la esquina superior izquierda aparezca la ventana a partir de las coordenadas indicadas por x e y a partir de Netscape 3, Internet Explorer 4 (JavaScript 1.1).
- **`scrollBy(x,y)`:** desplaza la ventana actual el número de pixels especificado por (x,y). A partir de Netscape 4 (JavaScript 1.2).
- **`scrollTo(x,y)`:** desplaza la ventana actual a las coordenadas especificadas por (x,y). A partir de Netscape 4 (JavaScript 1.2)

Veamos un pequeño ejemplo en el que vamos a mover la ventana actual (`moveTo`) o a redimensionarla (`resizeTo`) según los datos introducidos en los campos del formulario:

```
<html xmlns="http://www.w3.org/1999/xhtml">  
<head>  
<title>OBJETO WINDOW - redimensionar</title>  
<script type="text/javascript">
```

```

var x,y;

function MoverVentana() {
    x = document.FORMULARIO[0].value;
    y = document.FORMULARIO[0].value;
    window.moveTo(x,y);
}

function Redimensionar() {
    x = document.FORMULARIO[0].value;
    y = document.FORMULARIO[0].value;
    window.resizeTo(x,y) ;
}

</script>
</head>

<body>

<p align="center"><b> INTRODUCE las coordenadas x e y para mover o redimensionar la ventana:</b></p>

<center>

<table border="0" cellpadding="0" cellspacing="0">

<tr>

<td bgcolor="#FFFF96">

<form name="FORMULARIO">

<p><b>Coordenada X:</b><input type="text" name="X" size="10"/></p>

<p><b>Coordenada Y:</b><input type="text" name="Y" size="10"/></p>

<input type="button" value="mover ventana" name="B1" onclick="MoverVentana()"/>

<input type="button" value="Redimensionar" name="B2" onclick="Redimensionar()"/>

</form>

</td>

</tr>

</table>

</center>

</body>

</html>

```

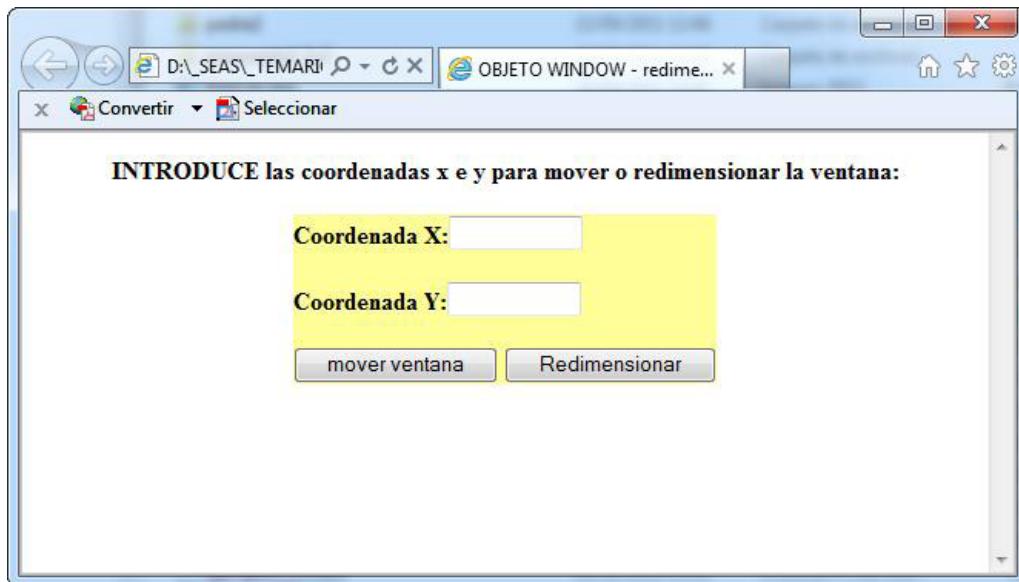


Figura 1.20. Salida ejemplo propiedades objeto Window – redimensionar.

Veamos más métodos del objeto Window:

- **alert(mensaje):** muestra el mensaje ‘mensaje’ en un cuadro de diálogo.
- **blur():** elimina el foco de la ventana y dispara el evento onBlur, a partir de Netscape 3, Internet Explorer 4 (JS 1.1).
- **clearInterval(id):** cancela el intervalo de tiempo referenciado por ‘id’, establecido por el método setInterval(). A partir de Netscape 4, Internet Explorer 4 (JavaScript 1.2).
- **clearTimeout(nombre):** cancela el intervalo de tiempo referenciado por ‘nombre’, establecido por el método setTimeout().
- **confirm(mensaje):** muestra el ‘mensaje’ en un cuadro de diálogo y dos botones, uno de Aceptar y otro de Cancelar. Devuelve true si se pulsa aceptar y devuelve false si se pulsa cancelar.
- **focus():** captura el foco del ratón sobre y dispara el evento onFocus. A partir de Netscape 3, Internet Explorer 4 (JavaScript 1.1).
- **prompt(mensaje,respuesta):** muestra el ‘mensaje’ en un cuadro de diálogo que contiene una caja de texto con dos botones (Aceptar y Cancelar) en la cual podremos escribir una respuesta a lo que nos pregunte el ‘mensaje’. El parámetro ‘respuesta’ es opcional, e inicialmente aparece en el campo de texto. Devuelve una cadena de caracteres con la respuesta introducida.

- ***setInterval(expresión,tiempo)***: evalúa la expresión cada vez que transcurren los milisegundos indicados en el segundo parámetro. Devuelve un valor que puede ser usado como identificativo por *clearInterval()*. A partir de Netscape 4, Internet Explorer 4 (JavaScript 1.2).
- ***setTimeout(expresión,tiempo)***: evalúa la expresión después de que hayan pasado el número de milisegundos especificados en tiempo. Devuelve un valor que puede ser usado como identificativo por *clearTimeout()*. A partir de Netscape 4, Internet Explorer 4 (JavaScript 1.2).

Vamos a ver ahora un ejemplo que creará una barra de estado móvil en la ventana del navegador. Al cerrar la ventana se visualizará un mensaje de despedida:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>OBJETO WINDOW - scroll</title>
<script type="text/javascript">

    Texto = " Ejemplo de Barra scroll ";
    i = 0;

    function MensajeScroll() {

        window.status = Texto.substring(i,Texto.length) +
Texto.substring(0,i-1);

        if (i < Texto.length) i++;
        else i = 0;
        setTimeout("MensajeScroll()",100);
    }
</script>
</head>
<body onload="MensajeScroll()" onunload="alert('Adiós')" >
    <b>MIRA EN LA BARRA DE ESTADO</b>
</body>
</html>
```

Si ejecutamos veremos la siguiente ventana:

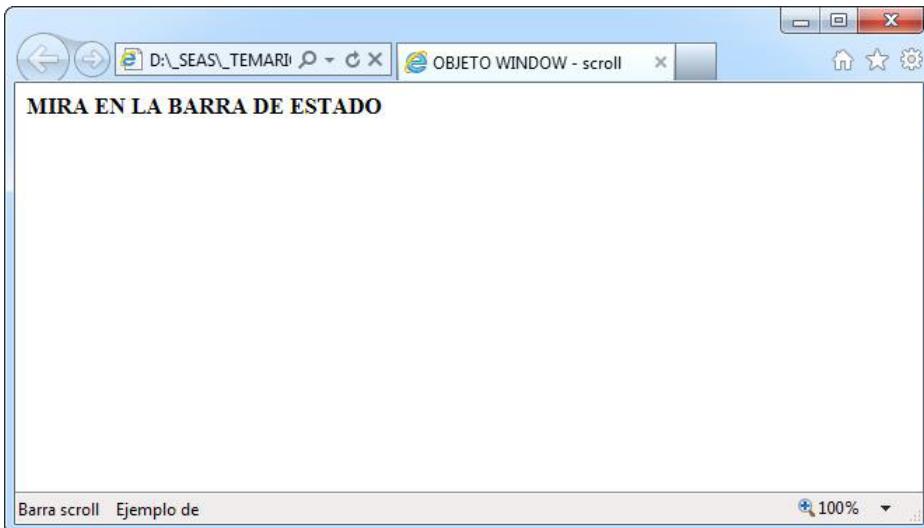


Figura 1.21. Ejemplo de barra scroll en movimiento.

En este ejemplo hemos utilizado métodos del objeto *String* que trabaja con cadenas de caracteres, *substring* que obtiene parte de una cadena y *length* que obtiene la longitud de una cadena (se verán más adelante). Se ha utilizado el evento *onload* que se dispara inmediatamente después de que todos los objetos del documento se han transferido al navegador.

Vamos a añadir al ejemplo anterior un botón, de forma que al hacer clic en él se imprima el contenido del documento que está visible en la ventana.

Añadimos la función imprimir con el método *print()*:

```
function Imprimir() {  
    window.print();  
}
```

Y el código asociado al botón dentro de la etiqueta *<body> </body>*:

```
<b>MIRA EN LA BARRA DE ESTADO</b>  
  
<form>  
    <p><input type="button" value="Imprimir"  
           name="imprimir" onclick="Imprimir()"/></p>  
</form>
```

Si ejecutamos veremos la siguiente salida:

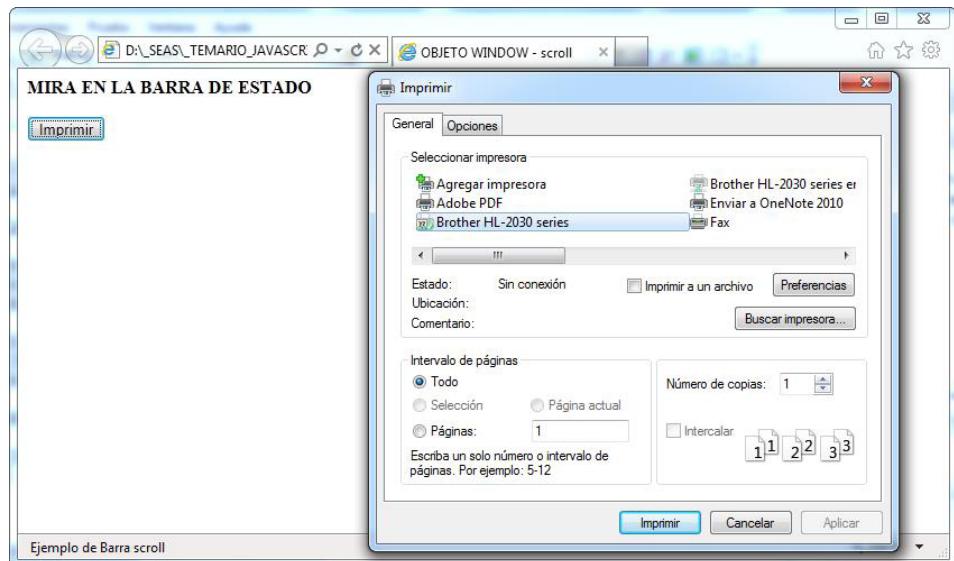


Figura 1.22. Vista de la ejecución del código de imprimir.

Seguimos con el ejemplo anterior, ahora antes de imprimir vamos a sacar un cuadro de diálogo pidiéndonos confirmación para imprimir. Usaremos el método `confirm()`. Ahora la función `Imprimir()` nos quedaría:

```
function Imprimir() {
    if (confirm("¿Tiene encendida la impresora?"))
        window.print();
}
```

Si el usuario pulsa el botón Aceptar, `confirm()` devuelve `true` y se ejecuta la sentencia para imprimir.

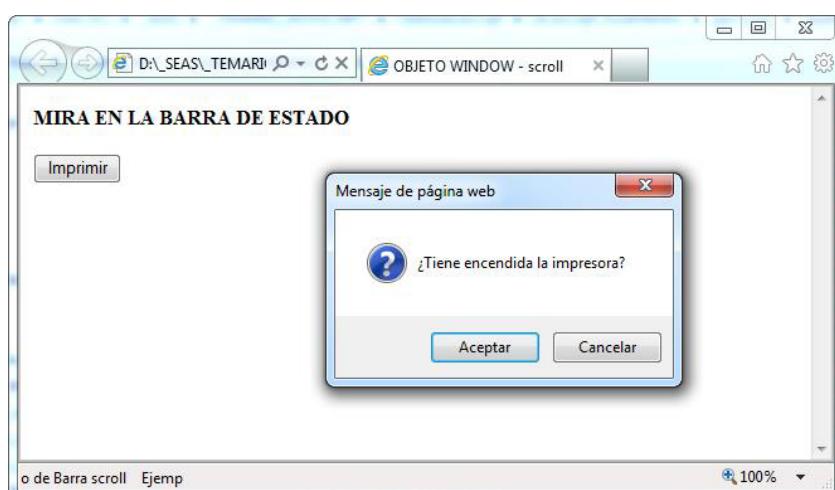


Figura 1.23. Vista del diálogo de confirmación antes de imprimir.

Vamos a seguir con ejemplos del objeto *Window*. El siguiente define varios hipervínculos de forma que al pasar el puntero del ratón por cada hipervínculo se visualiza un mensaje en la barra de estado. Al abandonarlo se dejará la barra de estado como estaba. El código para los hipervínculos es:

```
<body onunload="alert('Adiós')">

<b>MIRA EN LA BARRA DE ESTADO AL PASAR POR LOS HIPERVÍNCULOS.</b>

<p><b>Para visitar la pagina de Seas haz clic

<a href="http://estudiosabiertos.es"
    onmouseover="window.status='Ir a Estudios
Abiertos';return true;" onmouseout="window.status='';return true;">
aquí </a>, Para visitar la pagina del Campus Virtual haz clic

<a href="http://campus.seas.es/"
    onmouseover="window.status='Acceder a Campus Virtual'; re-
turn true;" onmouseout="window.status='';return true;">aquí </a></b>

</p>.

</body>
```

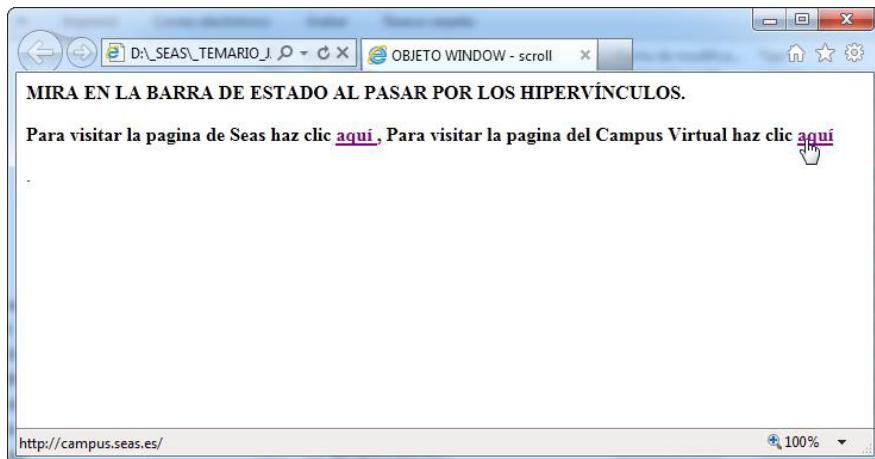


Figura 1.24. Vista de la ventana del script.

El evento `onmouseover` se dispara cuando el usuario pasa el puntero del ratón por el hipervínculo, `onmouseout` se dispara al salir del hipervínculo. Si la sentencia `return true;` no aparece, el mensaje no se visualiza en la línea de estado.

Y por último, el siguiente ejemplo crea una ventana hija donde se carga el documento. En este documento aparecerán una serie de hipervínculos a páginas externas. Al hacer clic en estos hipervínculos se cargarán en la ventana padre inicial (para realizar este ejemplo debemos crear dos documentos htm y guardarlos en la misma localización):

Código de la página HTMLPage1:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>OBJETO WINDOW - Ventana Padre </title>

<script type="text/javascript">

ventanahija = window.open("HTMLPage2.htm", "VentanaHija",
"width=175,height=175")

</script>

</head>

<body>

<center>

<p><b>ESTA PÁGINA CREA UNA VENTANA CON UN INDICE A DIVERSOS
EJEMPLOS</b></p>

<p>LOS EJEMPLOS SE CARGARÁN EN ESTA VENTANA</p>

</center>

</body>

</html>
```

Código de la página HTMLPage2:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1" />

<title>OBJETO WINDOW - Ventana Hija </title>

<script type="text/javascript">

function ColocarUrl(direccion) {

opener.document.location = direccion;

}

</script>

</head>
```

```

<body>
    <center>
        <p><b>HAZ CLIC EN UN EJEMPLO</b></p>
        <a href="javascript:ColocarUrl('http://seas.es')">Seas</a><br>
        <a href="javascript:ColocarUrl('http://estudiosabiertos.es')">Estudios Abiertos</a><br>
        <a href="javascript:ColocarUrl('http://googlees') ">Google
    </a><br>
    </center>
</body>
</html>

```

Si ejecutamos el código nos da la siguiente salida:

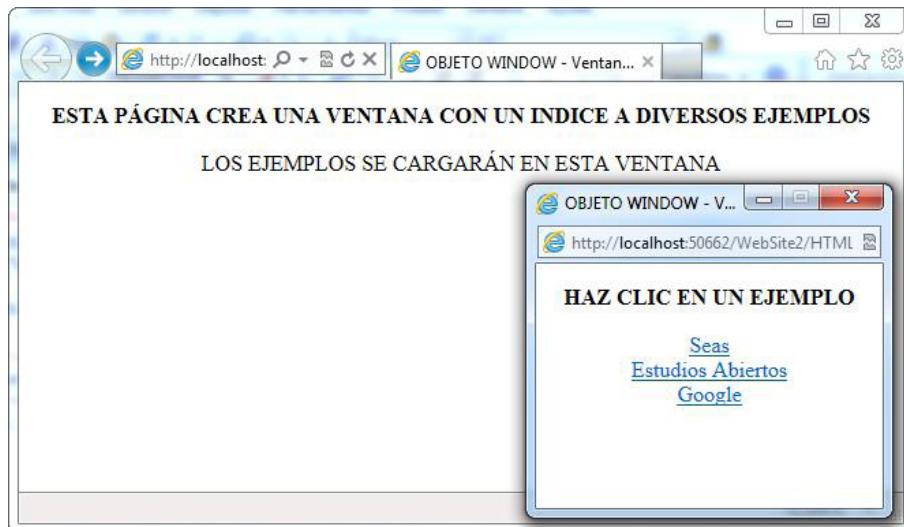


Figura 1.25. Vista de las ventanas del ejemplo anterior.

`opener` contiene la referencia a la ventana que abrió ésta, es el documento padre, con la sentencia `opener.document.location = direccion` se cargará en dicha ventana el documento indicado en `direccion`, también se podría haber puesto `opener.window.location`

Objeto Frame

Un objeto `window` puede contener múltiples subventanas denominadas *frames* o marcos. Cada marco puede contener un documento distinto, de esta forma una ventana puede contener diferentes documentos. Un objeto *frame* se comporta igual que un objeto `window`.

Las propiedades y métodos se resumen en la siguiente tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
defaultStatus	alert(), confirm(mensaje)	
window	clearInterval(), setInterval()	
frames	clearTimeout(), setTimeout()	
parent	focus(), blur(),	onLoad=
self	moveBy(x,y), moveTo(x,y)	onUnload=
status	open(), close()	
top	prompt(), scroll(x,y)	

Figura 1.26. Propiedades y métodos del objeto Frame.

Para acceder a las propiedades o métodos:

```
parent.nombreframe.propiedad  
parent.nombreframe.método()  
parent.frames[i].propiedad  
parent.frames[i].método()
```

Propiedades del objeto *Frame*:

- **frames:** array que representa los objetos *frame* que están contenidos en la ventana actual. El orden de aparición (frame[0],...) es el orden en que se definen en el documento HTML.
- **name:** nombre del objeto *frame*.
- **parent:** referencia al objeto *window* que contiene el *frame*.
- **self:** nombre alternativo del *frame*.
- **top:** nombre alternativo del objeto *window* padre de este *frame*.

Métodos del objeto *frame*:

La mayoría se han visto con el objeto *window*:

- **blur():** elimina el foco del frame y dispara el evento *onBlur*.
- **focus():** asigna el foco al frame y dispara el evento *onFocus*.

Un objeto *frame* siempre tiene una propiedad *top* y una propiedad *parent* diferente de su propiedad *self*.

El siguiente ejemplo define una página con dos marcos. El documento HTML que define el marco padre se llama `padre.htm`, los marcos hijos se llaman `frame1.htm` (situado en la parte superior) y `frame2.htm` (en la parte inferior). En el marco padre hemos dado nombre a la ventana con la sentencia `window.name='VENTANITA'`, los marcos hijos visualizan información referente al padre y a sí mismos. Veamos el código de `padre.htm`:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">  
<head>  
    <title>OBJETO FRAME - frames</title>  
    <script type="text/javascript">  
        window.name="VENTANITA";  
    </script>  
    <frameset rows="50%,50%">  
        <frame SRC="frame1.htm" name="FRAME1" noresize>  
        <frame SRC="frame2.htm" name="FRAME2" noresize>  
    </frameset>  
</head>  
<body>
```

El código de `frame1.htm` será el siguiente:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">  
<head>  
    <title>Página sin título</title>  
    <script type="text/javascript">  
        function infdocu()  
        {  
            document.write("Nombre de esta ventana: "+top.name +"  
");  
            document.write("Nombre del parent: "+parent.name+ "  
");  
            document.write("Mi nombre es: "+self.name+ "  
");  
            document.write("Nombre del frame de abajo: "+ top.frames[1].  
name +"  
");  
            document.write("URL de mi parent: "+ parent.location+ "  
");  
            document.write("Mi URL es: "+self.location+ "  
");  
        }  
    </script>
```

```

</script>

</head>

<body>

<script type="text/javascript">

infdocu();

</script>

</body>

</html>

```

El código es similar al del `frame1.htm`, solo varía la línea que visualiza el nombre del frame de arriba:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>

<title>Página sin título</title>

<script type="text/javascript">

function infdocu()

{
    document.write("Nombre de esta ventana: "+top.name +"  
");
    document.write("Nombre del padre: "+parent.name+ "  
");
    document.write("Mi nombre es: "+self.name+ "  
");

    document.write("Nombre del frame de arriba: "+ top.frames[0].name +"  
");
    document.write("URL de mi padre: "+ parent.location+ "  
");
    document.write("Mi URL es: "+self.location+ "  
");

}

</script>

</head>

<body>

<script type="text/javascript">

infdocu();

</script>

</body>

</html>

```

Vista de la ejecución del código:

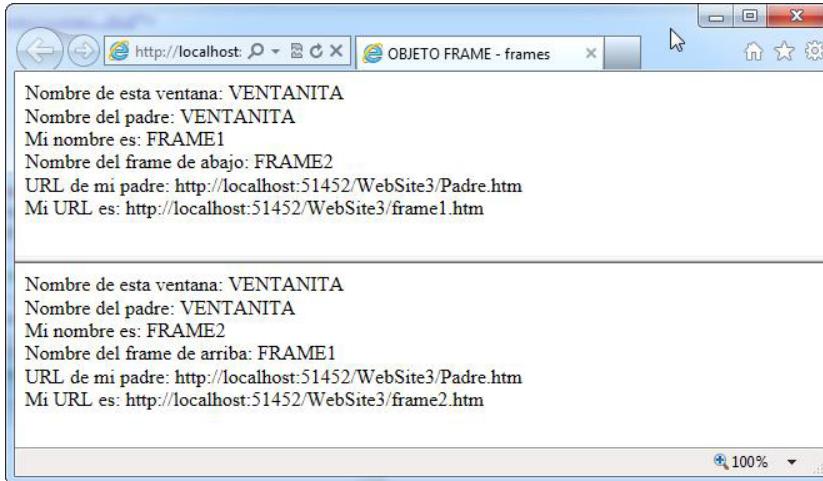


Figura 1.27. Vista de la ventana objeto Frame.

- ***self.parent***: hace referencia a la ventana que contiene el *frame* actual.
- ***self.parent.name***: obtiene el nombre del padre del *frame* actual.
- ***self.name***: hace referencia al nombre del *frame* actual.
- ***self.location***: obtiene la URL del *frame* actual.
- ***parent.location***: obtiene la URL del padre del *frame* actual.
- Todos los objetos *window* y *frame* tienen asociado un objeto *location*.

Vamos a ver otro ejemplo para entender mejor los *frames*. Supongamos que el *frame* de arriba (*arriba.htm*) es a la vez otro documento HTML que contiene dos marcos, uno a la derecha (*derecha.htm*), y otro a la izquierda (*izquierda.htm*). Al *frame* principal lo llamamos *padre2.htm*. Gráficamente lo podemos representar de la siguiente forma:

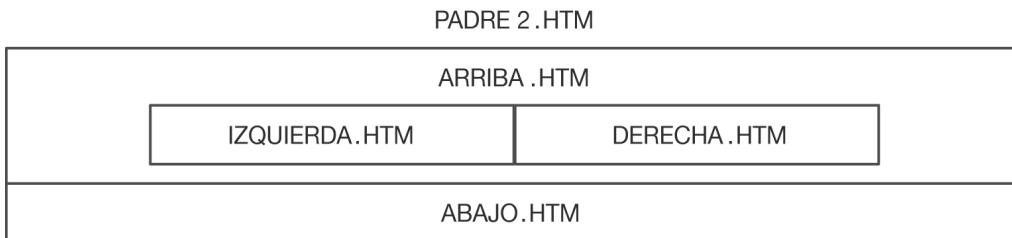


Figura 1.28. Cuadro resumen.

En la tabla siguiente mostramos los documentos padre, es decir, los que contienen los *frames*, que son *frame2.htm* y *arriba.htm*. Los nombres de padre de cada *frame* son PADRE2 y ARRIBA respectivamente.

```
padre2.htm
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD
XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/
xhtml">

<head>
    <title>OBJETO FRAME - frames2</title>
    <script type="text/javascript">
        {
            window.name="PADRE2";
        }
    </script>
</head>

<frameset rows="50%,50%">
    <frame src="arriba.htm"
name="ARRIBA">
    <frame src="abajo.htm"
name="ABAJO">
</frameset>

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

```
arriba.htm
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD
XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">

<html xmlns="http://www.
w3.org/1999/xhtml">

<head>
    <title>Página sin título</title>
</head>

<frameset cols="50%,50%">
    <frame src="izquierda.htm"
name="IZQUIERDA" ></frame>
    <frame src="derecha.htm"
name="DERECHA" ></frame>
</frameset>

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

Figura 1.29. Documentos padre que contienen los frames.

Veamos en los siguientes listados de los documentos HTML las órdenes que hacen referencia al nombre de los *frames*. En los documentos *izquierda.htm* y *derecha.htm*, para visualizar el nombre del *frame* de la derecha y el de la izquierda, respectivamente, se utiliza un *array* de dos dimensiones.

En el documento *izquierda.htm* visualizamos el nombre del *frame* de la derecha con *top.frames[0][1].name*, así cargamos el primer *frame* con posición 0 (ARRIBA) y el segundo *frame* dentro de este *frame*, posición 1 (DERECHA).

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript">
    function infdocu()
    {
        document.write("nombre de esta ventana:"+top.name+"<br>");
        document.write("nombre del parent:"+self.parent.name+"<br>");
        document.write("mi nombre es:"+self.name+"<br>");
        document.write("nombre del frame de abajo:"+top.frames[1].
name+"<br>");
        document.write("nombre del frame de arriba:"+top.frames[0].
name+"<br>");
        document.write("nombre del frame de la derecha:"+top.frames[0]
[1].name+"<br>");
    }
</script>
</head>
<body>
<script type="text/javascript">
    infdocu();
</script>
</body>
</html>

```

En el documento `derecha.htm` para visualizar el nombre del *frame* de la izquierda usamos `top.frames[0][0].name`, de esta manera el primer *frame* que se carga, es posición 0 (ARRIBA), y el primer *frame* dentro de este *frame*, posición 0 (IZQUIERDA).

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript">
    function infdocu()
    {
        document.write("nombre de esta ventana:"+top.name+"<br>");
        document.write("nombre del parent:"+self.parent.name+"<br>");
    }
</script>

```

```
        document.write("mi nombre es:"+self.name+"  
");
        document.write("nombre del frame de abajo:"+top.frames[1].
name+"  
");
        document.write("nombre del frame de arriba:"+top.frames[0].
name+"  
");
        document.write("nombre frame de la izquierda:"+top.frames[0]
[0].name+"  
");
    }
</script>
</head>
<body>
<script type="text/javascript">
    infdocu();
</script>
</body>
</html>
```

El documento `abajo.htm` será el siguiente:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script type="text/javascript">
    function infdocu()
    {
        document.write("nombre de esta ventana:"+top.name+"  
");
        document.write("nombre del padre:"+parent.name+"  
");
        document.write("mi nombre es:"+self.name+"  
");
        document.write("nombre del frame de arriba:"+top.frames[0].
name+"  
");
        document.write("URL de mi padre:"+parent.location+"  
");
        document.write("mi URL es:"+self.location+"  
");
    }
</script>
</head>
<body>
```

```

<script type="text/javascript">
infdocu();
</script>
</body>
</html>

```

Cargamos en el navegador padre2.htm y veremos la siguiente ventana:

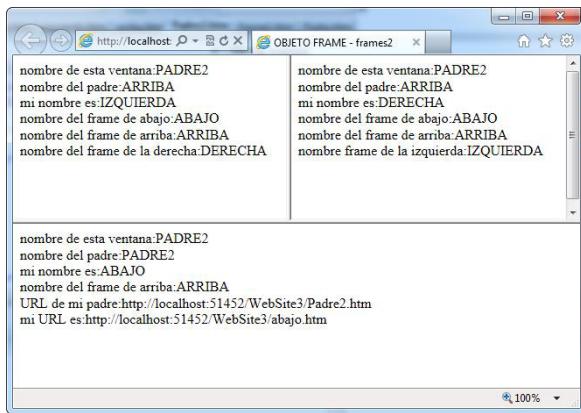


Figura 1.30. Vista de padre2.htm.

Objeto Location

Representa la información de la URL de cualquier ventana que esté actualmente abierta. Ya hemos visto en ejemplos anteriores cómo puede obtenerse la URL de un documento, y de un documento contenido en *frames*.

Las propiedades y métodos se resumen en esta tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
hash		
host		
hostname		
href	reload() replace(cadenaURL)	ninguno
pathname		
port		
protocol		
search		

Figura 1.31. Propiedades y métodos del objeto Location.

Para asignar a la ventana actual una nueva dirección escribiremos lo siguiente:

```
[window.]location="URL"
```

Para acceder a las propiedades o métodos:

```
[window.]location.propiedad
```

```
[window.]location.método()
```

La sentencia [window] es opcional

Supongamos que tenemos la siguiente dirección:

<http://www.servidor.com:80/camino/servicios.htm#primero>

Veamos cuales son las propiedades para la misma:

- **hash:** cadena que contiene el nombre del enlace, dentro de la URL. "#primero".
- **host:** contiene el nombre del servidor y el número del puerto: "www.servidor.com:80".
- **hostname:** contiene el nombre de dominio del servidor (o la dirección IP): "www.servidor.com"
- **href:** contiene la URL completa:
"http://www.servidor.com:80/camino/servicios.htm#primero".
- **pathname:** contiene el camino al recurso: "/camino/servicios.htm"
- **port:** cadena que contiene el número de puerto del servidor: "80"
- **protocol:** contiene el protocolo utilizado (incluyendo los dos puntos): ":80"
- **search:** mantiene la parte de URL que contiene la información que se pasa en una llamada a un script CGI.
- Los **métodos del Objeto Location** son los siguientes:
 - **reload():** vuelve a cargar el documento cuya URL se mantiene en la propiedad href.
 - **replace(cadenaURL):** carga el documento cuya URL se pasa como parámetro y hace que el historial actual sólo contenga esa dirección, no se podrá volver a la página anterior usando el botón Atrás del navegador.

Veamos un ejemplo de uso de estas propiedades. El siguiente script:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>OBJETO LOCATION - Ejemplo1</title>
```

```

<script type="text/javascript">
    function localizar() {
        document.write("HREF: " + location.href +"  
");
        document.write("HOST: " + location.host +"  
");
        document.write("HOSTNAME: " + location.hostname +"  
");
        document.write("PATHNAME: " + location.pathname +"  
");
        document.write("PORT: " + location.port +"  
");
        document.write("PROTOCOL: " + location.protocol +"  
");

    }
</script>
</head>
<body>
<center>
<p><b>ESTA PÁGINA ES UN EJEMPLO DEL OBJETO LOCATION</b></p>
<p>LOS EJEMPLOS SE VISUALIZARAN EN ESTA VENTANA</p>
<script type="text/javascript">
    localizar();
</script>
</center>
</body>
</html>

```

Obtendremos la siguiente salida:

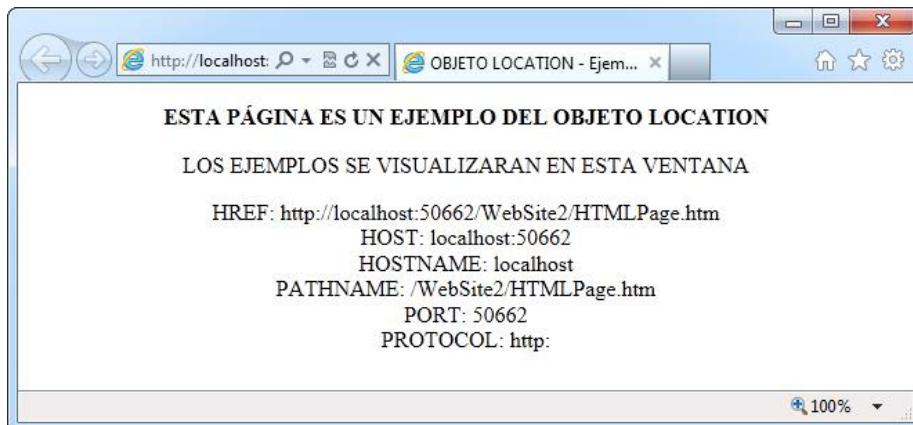


Figura 1.32. Vista de la ventana con el ejemplo.

Veamos otro ejemplo en el que utilizaremos el método `replace()`, al hacer clic en el botón se cargará la página de Seas, y no se podrá volver a la página anterior usando el botón Atrás del navegador:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>OBJETO LOCATION - Ejemplo2</title>
</head>
<body>
    <form>
        <p>
            <input type="button" value="Uso de replace"
                   onclick="location.replace('http://seas.es')"/></p>
        </form>
    </body>
</html>
```

Si lo ejecutamos nos aparecerá un botón que al pulsarlo nos cargará la página y no podremos volver atrás:

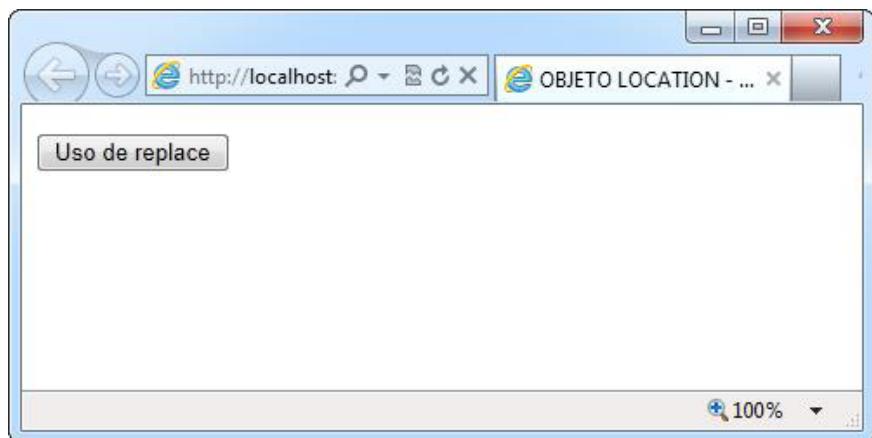


Figura 1.33. Vista de la ventana del ejemplo2.

Objeto History

Cuando navegamos por la red el explorador guarda una lista de las últimas URL donde se ha estado. Este objeto se encarga de representar esta lista y se utiliza, sobre todo, para movernos hacia delante o hacia atrás en dicha lista.

Las propiedades y métodos se resumen en este cuadro:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
length	back() forward() go(url)	ninguno

Figura 1.34. Propiedades y métodos del Objeto History

Para acceder a las propiedades o métodos:

```
[window.]history.propiedad
```

```
[window.]history.método()
```

La sentencia [window] es opcional.

Veamos las **propiedades del objeto History**:

- **length**: contiene el número de entradas de la lista del historial.

Y ahora los métodos del objeto History:

- **back()**: carga la URL de la entrada anterior del historial.
- **forward()**: carga la URL de la entrada siguiente.
- **go(url)**: carga la URL del documento especificado por ‘url’ del historial. La ‘url’ puede ser un entero, en cuyo caso indica la posición relativa del documento dentro del historial (si es mayor que 0 va hacia delante, si es menor que 0 hacia detrás y si es 0 vuelve a cargar la ventana); o puede ser una cadena de caracteres, en cuyo caso hace referencia a toda o parte de una URL que esté en el historial.

Veamos un ejemplo donde definimos dos botones para ir a la página anterior y a la siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>OBJETO HISTORY - Ejemplo1</title>
</head>
<body>
    <form>
        <p>
            <input type="button" value="Atras" onclick="history.back () "></p>
        <p>
            <input type="button" value="Adelante" onclick="history.forward () "></p>
        </form>
</body>
</html>
```

```

<input type="button" value="Siguiente" onclick="history.forward()">
</p>
</form>
</body>
</html>

```

Si ejecutamos el script observaremos la siguiente salida:

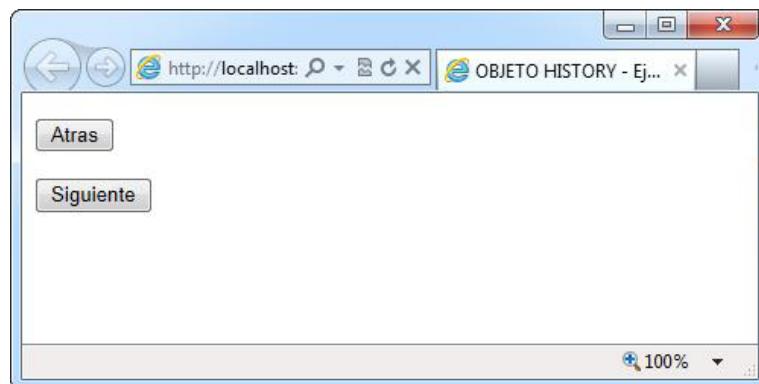


Figura 1.35. Vista del ejemplo1 del Objeto History.

Objeto Navigator

Este objeto no mantiene una relación directa con el resto de objetos del modelo de JavaScript, simplemente nos da información relativa al navegador que se esté utilizando para visualizar los documentos. Las propiedades y métodos se resumen en esta tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
appCodeName		
appName		
appVersion		
language		
mimeTypes	javaEnabled()	ninguno
platform		
plugins		
userAgent		

Figura 1.36. Propiedades y métodos del Objeto Navigator.

Para acceder a las propiedades o métodos:

navigator.propiedad

navigator.método()

Veamos las **propiedades del Objeto Navigator**:

- **appCodeName**: cadena que contiene el nombre del código del navegador del cliente.
- **appName**: contiene el nombre del navegador del cliente.
- **appVersion**: contiene la versión del navegador con el formato:
numerodeversión(plataforma;país)
- **language**: contiene información sobre el idioma de la versión del navegador.
- **mimeTypes**: array que contiene todos los tipos MIME soportados por el navegador del cliente.
- **platform**: contiene el tipo de máquina en la que se compiló el navegador.
- **plugins**: array que contiene todos los plug-ins soportados por el cliente.
- **userAgent**: contiene la cabecera del agente que se envía del cliente al servidor con el protocolo HTTP. Contiene la información de las propiedades `appCodeName` y `appVersion`.

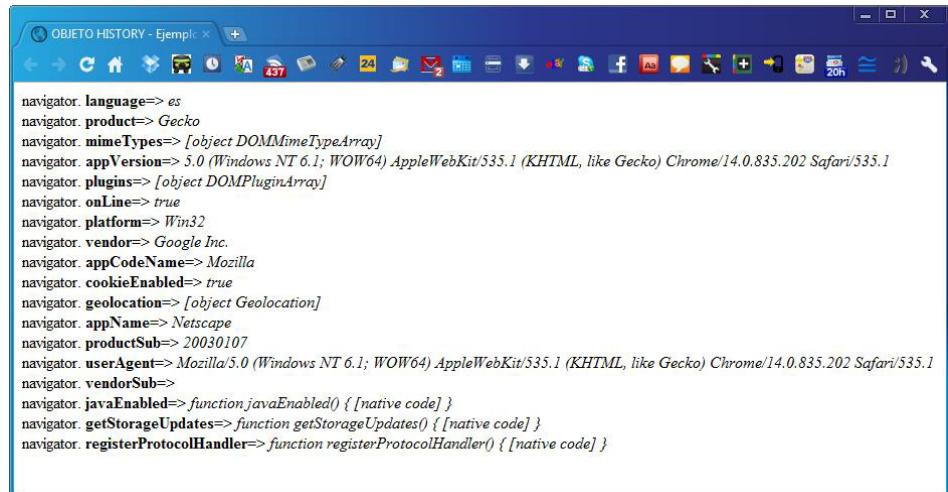
Los **métodos del Objeto Navigator** son los siguientes:

- **javaEnabled()**: devuelve ‘true’ si el cliente permite la utilización de Java, en caso contrario, devuelve ‘false’.

El siguiente código JavaScript muestra las propiedades del objeto **Navigator**:

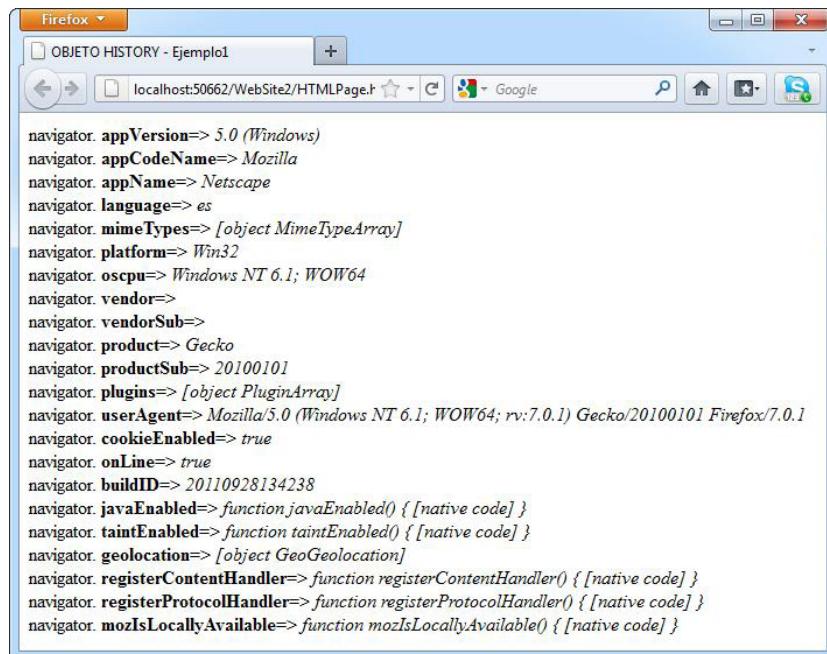
```
<script type="text/javascript">
    var i,navi;
    var navi=eval("navigator"); //eval evalúa el parámetro y devuelve el
    for (i in navi) // objeto asociado a la cadena
    {
        document.write("navigator. <b>" +i + "</b>=> <i>" + navi[i]
        + "</i><br>")
    }
</script>
```

Si ejecutamos el script en diferentes navegadores nos dará la siguiente salida:



```
navigator. language=> es
navigator. product=> Gecko
navigator. mimeTypes=> [object DOMMimeTypeArray]
navigator. appVersion=> 5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.835.202 Safari/535.1
navigator. plugins=> [object DOMPluginArray]
navigator. onLine=> true
navigator. platform=> Win32
navigator. vendor=> Google Inc.
navigator. appCodeName=> Mozilla
navigator. cookieEnabled=> true
navigator. geolocation=> [object Geolocation]
navigator. appName=> Netscape
navigator. productSub=> 20030107
navigator. userAgent=> Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.835.202 Safari/535.1
navigator. vendorSub=>
navigator. javaEnabled=> function javaEnabled() { [native code] }
navigator. getStorageUpdates=> function getStorageUpdates() { [native code] }
navigator. registerProtocolHandler=> function registerProtocolHandler() { [native code] }
```

Figura 1.37. Vista del script en Google Chrome.



```
navigator. appVersion=> 5.0 (Windows)
navigator. appCodeName=> Mozilla
navigator. appName=> Netscape
navigator. language=> es
navigator. mimeTypes=> [object MimeTypeArray]
navigator. platform=> Win32
navigator. oscpu=> Windows NT 6.1; WOW64
navigator. vendor=>
navigator. vendorSub=>
navigator. product=> Gecko
navigator. productSub=> 20100101
navigator. plugins=> [object PluginArray]
navigator. userAgent=> Mozilla/5.0 (Windows NT 6.1; WOW64; rv:7.0.1) Gecko/20100101 Firefox/7.0.1
navigator. cookieEnabled=> true
navigator. onLine=> true
navigator. buildID=> 20110928134238
navigator. javaEnabled=> function javaEnabled() { [native code] }
navigator. taintEnabled=> function taintEnabled() { [native code] }
navigator. geolocation=> [object GeoGeolocation]
navigator. registerContentHandler=> function registerContentHandler() { [native code] }
navigator. registerProtocolHandler=> function registerProtocolHandler() { [native code] }
navigator. mozIsLocallyAvailable=> function mozIsLocallyAvailable() { [native code] }
```

Figura 1.38. Vista del script en Internet Explorer.

```

navigator. appVersion=> 5.0 (Windows)
navigator. appCodeName=> Mozilla
navigator. appName=> Netscape
navigator. language=> es
navigator. mimeTypes=> [object MimeTypeArray]
navigator. platform=> Win32
navigator. oscpu=> Windows NT 6.1; WOW64
navigator. vendor=>
navigator. vendorSub=>
navigator. product=> Gecko
navigator. productSub=> 20100101
navigator. plugins=> [object PluginArray]
navigator. userAgent=> Mozilla/5.0 (Windows NT 6.1; WOW64; rv:7.0.1) Gecko/20100101 Firefox/7.0.1
navigator. cookieEnabled=> true
navigator. onLine=> true
navigator. buildID=> 20110928134238
navigator. javaEnabled=> function javaEnabled() { [native code] }
navigator. taintEnabled=> function taintEnabled() { [native code] }
navigator. geolocation=> [object GeoGeolocation]
navigator. registerContentHandler=> function registerContentHandler() { [native code] }
navigator. registerProtocolHandler=> function registerProtocolHandler() { [native code] }
navigator. mozIsLocallyAvailable=> function mozIsLocallyAvailable() { [native code] }

```

Figura 1.39. Vista del script en Firefox.

1.10.3. Propiedades y métodos de los objetos del documento

Fijémonos en el gráfico que muestra la jerarquía de objetos de *JavaScript* en la figura 1.6, vemos que el objeto *Document* abarca gran parte del modelo de objetos. La mayor parte de comunicación entre el script y el usuario se realiza a través de este objeto y sus elementos, por ello es necesario comprender el campo de acción de dicho objeto.

1.10.3.1. Objeto Document

El objeto *Window* y el objeto *Frame* tienen asociado un objeto *Document* que es el que realmente contiene el resto de los objetos como son: texto, imágenes, enlaces, formularios, etc., que se muestran en el navegador.

En la siguiente tabla se muestran las propiedades y métodos más significativos:

PROPIEDADES	MÉTODOS
alinkColor, anchors, applets	clear()
bgColor, cookie, fgColor , forms	open()
images, lastModified	close()
linkColor, links location , referrer	write()
title ,vlinkColor	writeln()

Figura 1.40. Propiedades y métodos del objeto Document.

Para acceder a las propiedades y métodos del documento:

[window.]document.propiedad

[window.]document.método()

La sentencia [window] es opcional.

Las **propiedades del objeto Document** son las siguientes:

- **alinkColor:** almacena el color de los enlaces activos.
- **anchors:** array que contiene una entrada por cada enlace interno (etiqueta <A> con el atributo NAME) existente en el documento.
- **applets:** array con los applets existentes en el documento.
- **bgColor:** almacena el color de fondo del documento.
- **cookie:** cadena que contiene el valor de una cookie.
- **fgColor:** almacena el color del primer plano.
- **forms:** array que contiene una entrada por cada formulario definido en el documento.
- **images:** array con todas las imágenes del documento.
- **lastModified:** cadena con la fecha de la última modificación del documento.
- **linkColor:** color de los enlaces del documento.
- **links:** array que contiene una entrada por cada enlace externo existente en el documento (etiquetas <AREA REF.="" ...> y <A REF.="" >).
- **location:** cadena con la URL del documento actual.
- **referrer:** cadena con la URL del documento que llamó al actual, en caso de usar un enlace.

- **title:** título del documento actual.
- **vlinkColor:** almacena el color de los enlaces visitados

Veamos ahora los **métodos del objeto Document**:

- **open(*mime*, "replace"):** abre la escritura sobre el documento. 'mime' es el tipo de documento soportado por el navegador. Si se indica "replace", se reutiliza el documento anterior (no crea una nueva entrada en el historial).
- **close():** cierra la escritura sobre el documento actual y fuerza la visualización de su contenido.
- **write():** escribe texto en el documento.
- **writeln():** escribe texto en el documento y lo finaliza con un salto de línea.

Vamos a ver un ejemplo que nos mostrará las características del documento: número de enlaces, de formularios, de imágenes, color de fondo, fecha última modificación, etc. En él se definen tres enlaces, uno a otro documento HTML y los otros dos a un ancla o enlace dentro del documento, una imagen y un formulario:

```
<html xmlns="http://www.w3.org/1999/xhtml">

<head>

    <title>OBJETO DOCUMENT - Ejemplo1</title>

    <script type="text/javascript">

        var ventana;

        function CrearVentana() {
            var opciones="left=100,top=100,";
            opciones=opciones + "width=" + document.FORMULARIO[0].value + ",";
            opciones=opciones + "height=" + document.FORMULARIO[1].value ;
            ventana = window.open("", "", opciones); //crea la ventana
            ventana.document.write("ESTA ES LA VENTANA QUE SE HA CREADO");
        }

        function CerrarVentana() {
            if(ventana && !ventana.closed)
                ventana.close(); //cierra la ventana creada
            window.close(); //cierra la ventana actual
        }
    </script>
</head>
```

```

<body>
    <a href="#caracteristicas">Características</a>
    <p align="center">
        <b>INTRODUCE EL ANCHO Y ALTO DE LA VENTANA:</b></p>
    <p align="center">
        </p>
    <center>
        <table border="0" cellpadding="0" cellspacing="0">
            <tr>
                <td bgcolor="#FFFF66">
                    <form name="FORMULARIO">
                        <p>
                            <b>ANCHO VENTANA:</b><input type="text" name="ANCHO" size="10"></p>
                        <p>
                            <b>ALTO VENTANA:</b>
                            <input type="text" name="ALTO" size="10"></p>
                            <input type="button" value="Crear Ventana" name="B1" onclick="CrearVentana()">
                            <input type="button" value="Cerrar Ventana" name="B2" onclick="CerrarVentana()">
                        </form>
                    </td>
                </tr>
            </table>
        </center>
        <p align="center">
            <b><a href="http://www.seas.es">Enlace externo</a></b></p>
            <hr>
        <center>
            <b>CARACTERISTICAS DEL DOCUMENTO</b><br/>
            <a name="caracteristicas">características</a>
            <script type="text/javascript">
                document.write("Número de imágenes: " + document.images.length + " <br>");
                document.write("Número de enlaces externos: " + document.links.length + " <br>");
            </script>
        </center>
    </p>

```

```

document.write("Número de enlaces internos: " + document.an-
chors.length" <br>);

document.write("Nombre enlaces internos: " + document.
anchors[0].name +" <br>");

document.write("Número de formularios: " + document.forms.
length" <br>");

document.write("Color de fondo: " + document.bgColor+" <br>");

document.write("Color de los enlaces: " + document.linkColor+" 
<br>");

document.write("Color de los enlaces activos: " + document.
alinkColor+" <br>");

document.write("Fecha última modificación: " + document.lastMo-
dified+" <br>");

</script>

</center>

</body>

</html>

```

Si ejecutamos el script veremos la siguiente salida:

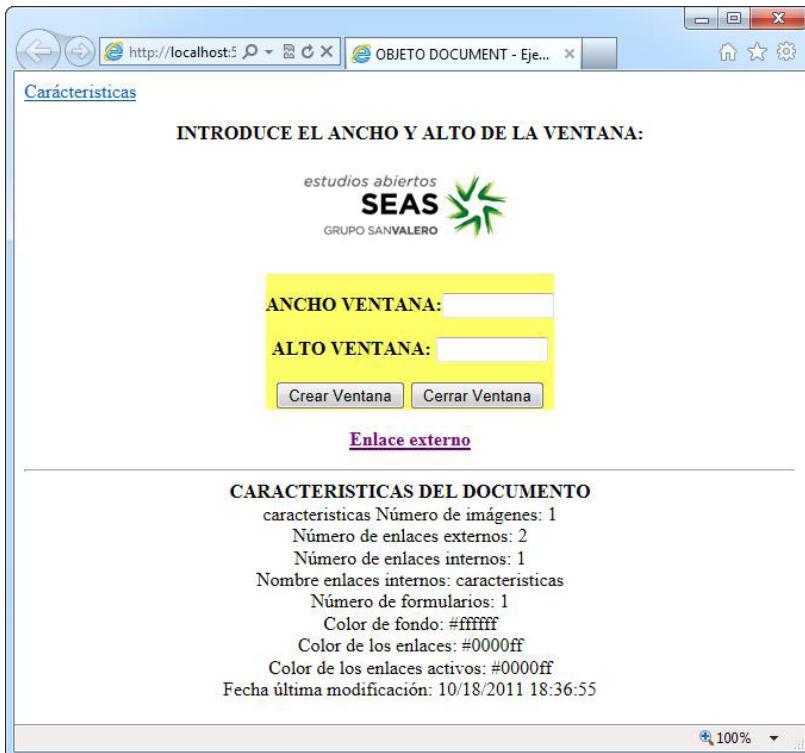


Figura 1.41. Vista del ejemplo1 del objeto Document.

1.10.3.2. Objetos Link y Anchor

Si el documento contiene algún hiperenlace local o externo, el objeto *Document* tendrá asociados distintos objetos *Link* y *Anchor*. El objeto *Link* engloba todas las propiedades que tienen los enlaces externos al documento actual y el objeto *Anchor* todas las propiedades que tienen los enlaces locales al documento actual. Las propiedades del objeto *Link* se resumen en la siguiente tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
links[indice].target		onClick=
length	ninguno	onMouseOver=
propiedades del objeto location		

Figura 1.42. Propiedades y métodos del objeto Link.

Veamos más especificadas sus propiedades:

- **target:** es una cadena que contiene el nombre de la ventana o del frame especificado en el parámetro TARGET. Para hacer referencia a cada uno de estos objetos y poder obtener el valor de sus propiedades usamos la expresión:

```
document.links[i].propiedad
```

Donde i es el enlace que se está explorando (0 es el primer enlace del documento, 1 el segundo...).

- **length:** para obtener el número de enlaces externos:

```
document.links.length
```

Para obtener el número de enlaces internos:

```
document.anchors.length
```

Partiendo del ejemplo anterior y cambiando el script:

```
<script type="text/javascript">
    var i;
    document.write("Número de enlaces externos: "+ 
    document.links.length+ " <br>"); 
    document.write("Número de enlaces internos: "+ 
    document.anchors.length+ " <br><hr>"); 
    for(i=0; i<document.links.length; i++) {
        document.write("Dirección href: "+ document.links[i].href+ "<br>"); 
        document.write("Nombre de enlace hash: "+document.links[i].hash+ "<br>"); 
    }
</script>
```

```

        + "<br>") ;

document.write("Servidor hostname: " + document.links[i].hostname
        + "<br>");

document.write("Protocol: " + document.links[i].protocol + "<br>");
document.write("Pathname: " + document.links[i].pathname + "<br><hr>");

}

</script>

```

Observaremos la siguiente salida:

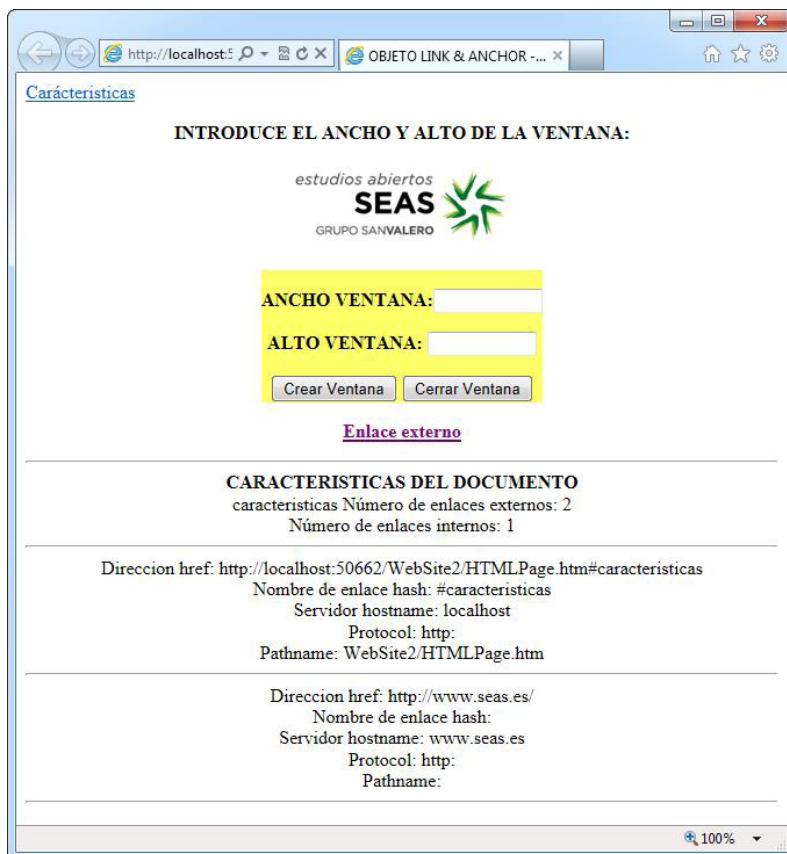


Figura 1.43. Vista del ejemplo1 del objeto Link.

1.10.3.3. Objeto Image

Este objeto nos permitirá manipular las imágenes del documento, pudiendo conseguir efectos como el conocido rollover o cambio de imágenes, por ejemplo, al pasar el ratón sobre la imagen. Las propiedades se resumen en la siguiente tabla:

PROPIEDADES	MÉTODOS
border, complete, height, hspace, lowsrc, name, src, vspace, width	Explorer y Mozilla no tienen ningún método común asociado a este objeto.

Figura 1.44. Propiedades y métodos del objeto Image.

Para acceder a las propiedades:

```
[window.]document.nombreImagen.propiedad
```

Veamos las **propiedades del objeto Image**:

- **border**: contiene el valor del atributo ‘border’ de la imagen.
- **complete**: Valor booleano que indica si la imagen se ha descargado completamente o no.
- **height**: contiene el valor del atributo ‘height’ de la imagen (alto).
- **hspace**: contiene el valor del atributo ‘hspace’ de la imagen (espacio horizontal alrededor de la imagen).
- **lowsrc**: contiene el valor del atributo ‘lowsrc’ de la imagen.
- **name**: contiene el valor del atributo ‘name’ de la imagen.
- **src**: contiene el valor del atributo ‘src’ de la imagen .
- **vspace**: contiene el valor del atributo ‘vspace’ de la imagen.
- **width**: contiene el valor del atributo ‘width’ de la imagen.

Vamos a realizar un ejemplo que muestra cómo crear imágenes de sustitución. En el documento aparecen dos imágenes (de nombres img1 e img2) que son dos hipervínculos, al pasar el ratón sobre ellas (evento *onMouseover*) la imagen cambia, cuando el ratón abandona la imagen (evento *onMouseout*) esta vuelve a cambiar. El código es el siguiente:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml">
```

```

<head>
    <title>OBJETO IMAGE - Ejemplo1</title>
</head>
<body>
    <center>
        <a href="http://seas.es"
            onmouseover="document.img1.src='img1.jpg'"
            onmouseout="document.img1.src='img2.jpg'>
            
        </a>
        <a href="http://www.opositio.es"
            onmouseover="document.img2.src='img2.jpg'"
            onmouseout="document.img2.src='img1.jpg'>
            
        </a>
    </center>
</body>
</html>

```

Si ejecutamos obtendremos la siguiente salida:



Figura 1.45. Vista del ejemplo1 del objeto Image.

Para que cambie la imagen al pasar el ratón por encima o al abandonarla cambiamos el valor de la propiedad src de dicha imagen:

```
document.nombreImagen.src='nuevaimagen'
```

También se podía haber puesto:

```
document['nombreImagen'].src='nuevaImagen'
```

Es decir:

```
document.img2.src='img1.jpg' ó document['img2'].src='img1.jpg'
```

1.10.3.4. Objetos del formulario

En este apartado veremos cómo manipular los objetos de un formulario, así podremos hacer funciones que nos permitan validarlos antes de enviar los datos a un servidor. *JavaScript* define un conjunto de objetos relacionados con los elementos que se pueden incluir dentro de un formulario, son los siguientes: *form*, *text*, *textarea*, *button*, *checkbox*, *radio*, *select*, *password* y *hidden*. Toda etiqueta <FORM> de HTML tiene asociada un objeto *Form* de *JavaScript*.

Las propiedades y métodos del objeto Form se resumen en esta tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
action, elements encoding, length , method target	reset() submit()	OnSubmit=

Figura 1.46. Propiedades y métodos del objeto Form.

Para acceder a las propiedades o métodos:

```
[window.]document.nombreformulario.propiedad
[window.]document.nombreformulario.método()
[window.]document.forms[indice].propiedad
[window.]document.forms[indice].método()
```

Si en el documento tenemos tres formularios podemos utilizar:

`document.forms[0]` para referenciar el primer formulario.

`document.forms[1]` para el segundo .

... para el tercero

Veamos las propiedades del objeto Form:

- **action:** cadena que contiene el valor del atributo ACTION del formulario (URL del programa que procesa los datos del formulario).

- **elements:** array que contiene todos los elementos del formulario, en el mismo orden en el que se definen en el documento HTML. Por ejemplo, si en el formulario hemos puesto, en este orden, una caja de texto, un checkbox y una lista de selección, la caja de texto será elements[0], el checkbox será elements[1] y la lista de selección será elements[2].
 - **encoding:** cadena que contiene el valor del atributo ENCTYPE del formulario (código MIME).
 - **length:** número de elementos que contiene el formulario.
 - **method:** cadena que contiene el valor del atributo METHOD del formulario (método con el que se va a recibir/procesar la información del formulario GET/POST).
 - **target:** cadena que contiene el valor del atributo TARGET (nombre de ventana o marco donde aparecerá la respuesta que genere el servidor después de enviar el formulario).
- Y los **métodos del objeto Form** son los siguientes:
- **reset():** simula el mismo efecto que si pulsáramos un botón de tipo RESET dispuesto en el formulario (borra los datos que haya introducido el usuario en el formulario).
 - **submit():** envía el formulario. Tiene el mismo efecto que si pulsáramos un botón de tipo SUBMIT dispuesto en el formulario.

Recordemos la sintaxis para crear un formulario:

```
<form name="NombreFormulario"  
      [target="NombreVentana"]  
      [action="URLservidor"]  
      [method= "get|post"  
      [enctype="TipoMIME"  
      [onsubmit="Función_texto"] >  
      ...objetos del formulario...  
</form>
```

Si el formulario no va a tener peticiones o envíos a través del servidor los atributos **action=**, **target=** y **method=** no son necesarios. Pero si el formulario va a realizar peticiones o preguntas al servidor, será necesario especificar al menos los atributos **action** y **method=**; especificaremos el atributo **target=** si los datos resultantes del servidor se van a mostrar en otra ventana que no sea la que realiza la llamada. Igualmente especificaremos el atributo **enctype=** si los script de formulario dan forma a los datos ligados al servidor en un tipo **MIME=** que no sea un flujo ASCII.

En este apartado aprenderemos a utilizar *JavaScript* para validar los datos del formulario, es decir, para asegurarnos que los formularios contienen información válida antes de que se envíen al servidor.

Casi todas las etiquetas HTML utilizadas para definir elementos en un formulario tienen asociado un objeto *JavaScript*. La siguiente tabla muestra la relación entre los elementos HTML de un formulario y los objetos de *JavaScript*:

Etiqueta HTML	Objeto JavaScript
<INPUT type="text">	text
<TEXTAREA>	textarea
<INPUT type="password">	password
<INPUT type="button">	button
<INPUT type="submit">	submit
<INPUT type="reset">	reset
<INPUT type="checkbox">	checkbox
<INPUT type="radio">	radio
<SELECT>	select
<INPUT type="hidden">	hidden

Figura 1.47. Relación entre etiquetas HTML y los objetos de JavaScript.

A continuación vamos a ver las propiedades y métodos de estos objetos.

Objetos Text, Textarea y Password

Estos objetos son el medio principal para la obtención de texto introducido por el usuario. Representan los campos de texto y las áreas de texto dentro de un formulario. El objeto *Password* es igual que el objeto *Text* salvo que los caracteres introducidos por el usuario se muestran poniendo asteriscos (*) en su lugar. El objeto *Textarea* está destinado para la introducción de múltiples líneas de texto. Los tres tienen las mismas propiedades y métodos, por ello los vemos juntos.

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
defaultValue	blur()	onBlur=
name	focus()	onFocus=
value	select()	onChange=
		onSelect=

Figura 1.48. Propiedades y métodos de objetos Text, Textarea y Password.

Para acceder a las propiedades o métodos:

```
[window.]document.nombreformulario.nombrecampo.propiedad  
[window.]document.nombreformulario.nombrecampo.método()  
[window.]document.nombreformulario.elements[n].propiedad  
[window.]document.nombreformulario.elements[n].método()  
[window.]document.forms[m].nombrecampo.propiedad  
[window.]document.forms[m].nombrecampo.método()  
[window.]document.forms[m].elements[n].propiedad  
[window.]document.forms[m].elements[n].método()
```

Veamos ahora las propiedades de los objetos text, textarea y password:

- **defaultValue**: cadena que contiene el valor por defecto dado al objeto (valor del parámetro VALUE).
- **name**: cadena que contiene nombre del campo (valor del parámetro NAME).
- **value**: cadena que contiene el valor actual del campo.

Los **métodos de los objetos text, textarea y password** son los siguientes:

- **blur()**: elimina el foco del objeto.
- **focus()**: asigna el foco del ratón al objeto.
- **select()**: selecciona el texto que contiene el objeto

Vamos a ver ahora un ejemplo en el que realizaremos un formulario, en el cual pediremos al usuario su nombre y la contraseña y de nuevo la contraseña para validarla. Si las contraseñas no coinciden se visualizará un cuadro de diálogo indicándolo, entonces el foco de la ejecución volverá al campo contraseña (método `focus()`) que además aparecerá seleccionado (método `select()`). Si no se teclea nada en el campo contraseña también aparecerá un mensaje y el foco retornará a dicho campo. Si los datos han sido validados se enviarán al servidor (método `submit()`) a un programa ficticio SEAS. El código es el siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml">  
<head>  
    <title>OBJETO TEXT, TEXTAREA y PASSWORD - Ejemplo1</title>  
    <script type="text/javascript">  
        function validarFormulario(formulario) {  
            if (formulario.clave1.value == "") {  
                //no se ha introducido contraseña  
                alert("Debes introducir la contraseña");  
                formulario.clave1.focus(); //devolver el foco al campo  
                return;  
            }  
        }  
    </script>  
</head>  
<body>  
    <form name="miFormulario" onsubmit="return validarFormulario(this);>  
        <table border="1">  
            <tr>  
                <td>Nombre:</td>  
                <td>  
                    <input type="text" name="nombre" value="Juan" />  
                </td>  
            </tr>  
            <tr>  
                <td>Contraseña:</td>  
                <td>  
                    <input type="password" name="clave1" value="" />  
                </td>  
            </tr>  
            <tr>  
                <td>Repetir Contraseña:</td>  
                <td>  
                    <input type="password" name="clave2" value="" />  
                </td>  
            </tr>  
            <tr>  
                <td colspan="2" style="text-align: center;">  
                    <input type="submit" value="Enviar" />  
                </td>  
            </tr>  
        </table>  
    </form>  
</body>
```

```

        }else{
            if (formulario.clave1.value != formulario.clave2.value)
            {
                /las contraseñas escritas no coinciden
                alert("Las contraseñas introducidas son distintas")
                formulario.clave1.focus(); //devolver el foco
                formulario.clave1.select(); //seleccionar texto
                return;
            }else{
                formulario.action="http://www.servidor.es/login_servidor.seas"
                formulario.submit(); //enviar datos al servidor
            }
        }
    }
</script>
</head>
<body>
<p align="center">
<b>VERIFICACIÓN DE LA CONTRASEÑA</b>
<center>
<table bgcolor="#FFFFCC" border="2">
<td>
<form onsubmit=javascript:validarFormulario(this)
method="post">
Escribe tu nombre:
<input size="30" name="NOMBRE"/>
<p>
Contraseña:
<input name="clave1" type="password" size="20"/>
<p>
Introduce de nuevo la contraseña:
<input name="clave2" type="password" size="20"/>
<p>
<center>
<input type="submit" value="Enviar" name="Enviar"/>
nbsp;
<input type="reset" value="Restablecer"/>
</center>
</form>
</td>

```

```

    </table>
</center>
</p>
</body>
</html>

```



Figura 1.49. Vista del ejemplo en el navegador.

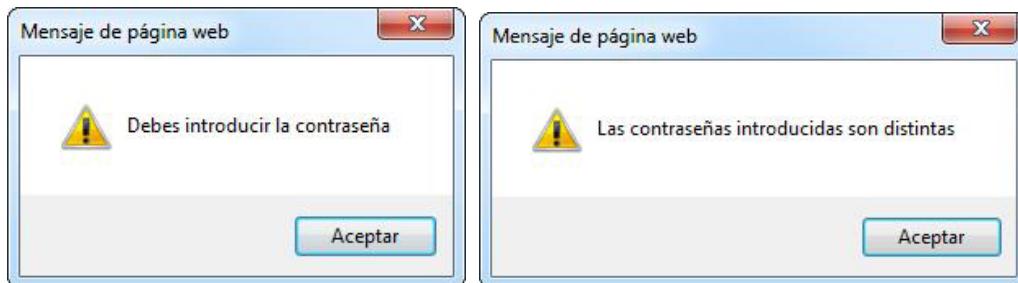


Figura 1.50. Mensajes de error cuando se introducen datos erróneos.

El evento `onsubmit` se dispara cuando un formulario está a punto de enviarse. Cuando se envíe el formulario (se hace clic en el botón “Enviar”, `type="submit"`) se invoca a la función `validarFormulario()`, el parámetro que se envía es `this` que hace referencia al objeto actual, es decir, al formulario. Se ha definido el método POST para enviar la información al servidor, `method="post"`. Cuando el formulario haya sido validado se realizan las siguientes acciones:

```

formulario.action="http://www.servidor.es/login_servidor.cgi"

formulario.submit(); //enviar datos al servidor

```

Es decir la acción a realizar será enviar los datos al servidor, a un programa ficticio “SEAS” que se encargará de procesar la información; a la URL indicada por el atributo `action`. Este formulario es un ejemplo de cómo enviar datos al servidor para ser procesados por un archivo.

Objetos button, reset y submit

Definen los tres tipos de botones que se pueden incluir en un formulario. Son los siguientes:

- Un botón genérico, “button”, que no tiene acción asignada.
- Un botón “submit”: al ser pulsado envía los datos del formulario.
- Un botón “reset”: al ser pulsado limpia los valores del formulario.

Las propiedades y métodos se resumen en esta tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
name		
value	click()	onClick=

Figura 1.51. Propiedades y métodos de los objetos Button, Reset y Submit.

Para acceder a las propiedades o métodos se hace de forma similar a los objetos vistos anteriormente. Veamos ahora las **propiedades de los objetos Button, Reset y Submit**:

- `name`: cadena que contiene el valor del parámetro NAME.
- `value`: cadena que contiene el valor del parámetro VALUE.

Y ahora veamos los métodos de los objetos Button, Reset y Submit:

- `click()`: reproduce la acción que el usuario realiza cuando hace clic en un botón.

Objeto Checkbox

Las casillas de verificación o *checkboxes* nos permiten seleccionar varias opciones marcando el cuadrito que aparece a su izquierda. La marca de la casilla equivale a un valor `true` y si no está marcada equivale a un valor `false`. Las propiedades y métodos se resumen en esta tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
checked		
defaultChecked		
name		
value	click()	onClick=

Figura 1.52. Propiedades y métodos del objeto Checkbox.

Propiedades del objeto Checkbox:

- **checked:** valor booleano que indica si la checkbox está seleccionada (true) o no seleccionada (false).
- **defaultChecked:** valor booleano que indica si la checkbox debe estar seleccionado por defecto o no.
- **name:** cadena que contiene el valor del parámetro NAME.
- **value:** cadena que contiene el valor del parámetro VALUE.
- y los **métodos de los objetos Checkbox:**
- **click():** reproduce la acción que el usuario realiza cuando hace clic en un botón.

Veamos un ejemplo en el que vamos a realizar un formulario en el que pediremos al usuario que introduzca su nombre y seleccione las casillas deseadas. Los botones de opción no tienen funcionalidad en este ejemplo. Al hacer clic en el botón “Calcular”, se visualizará en el campo TOTAL un total de euros que dependerá de las casillas seleccionadas:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>OBJETO CHECKBOX - Ejemplo1</title>
    <script type="text/javascript">
        function Calculo() {
            var tot=0;
            if (document.FORMULARIO.AIRE.checked) tot=tot+120;
            if (document.FORMULARIO.ELEVACONDICIONADO.checked) tot=tot+50;
            if (document.FORMULARIO.RADIO.checked) tot=tot+40;
            document.FORMULARIO.TOTAL.value=tot;
        }
    </script>
</head>
<body bgcolor="#eeee00">
    <form name="FORMULARIO" method="post" action="">
        <p>
            <b>accesorios</b><b>:</b>
            <b><input type="checkbox" name="AIRE"/></b>aire
            acondicionado (120€)
            <b><input type="checkbox" name="ELEVACONDICIONADO"/></b>
            elevacondicionado (50€)
            <b><input type="checkbox" name="RADIO"/></b>radio
            cd (40€) </p>
        <center>
```

```

<input type="button" value="calcular"
name="calcular" onclick="Calculo()"/>

<b>TOTAL:</b>

<input type="text" name="TOTAL" size="10" va-
lue="0" readonly="readonly"/>Euros

</center>

</form>

</body>

</html>

```

Si ejecutamos observaremos la siguiente salida:

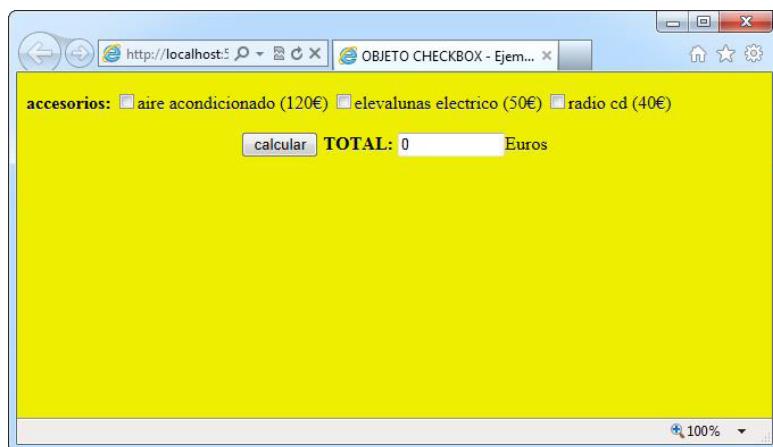


Figura 1.53. Vista del ejemplo1 del objeto Checkbox.

Objeto Ratio

Este objeto nos permitirá elegir una posibilidad entre todas las que hay. Todos los botones de un grupo van a compartir el mismo nombre, de esta manera JavaScript conoce al grupo de botones de tal forma que al hacer clic en uno de ellos se desactive el resto de botones del grupo.

Las propiedades y métodos se resumen en esta tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
checked		
defaultChecked		
length	click()	
name		onClick=
value		

Figura 1.54. Propiedades y métodos del objeto Ratio.

Para acceder a sus propiedades y métodos lo haremos de la siguiente manera:

```
[window.]document.nombreformulario.grupobotones[x].propiedad  
[window.]document.nombreformulario.grupobotones[x].método()  
[window.]document.nombreformulario.elements[n].propiedad  
[window.]document.nombreformulario.elements[n].método()  
[window.]document.forms[m].grupobotones.propiedad  
[window.]document.forms[m].grupobotones.método()  
[window.]document.forms[m].elements[n].propiedad  
[window.]document.forms[m].elements[n].método()
```

Veamos cada una de las propiedades del objeto Ratio:

- **checked:** valor booleano que nos dice si el radio está seleccionado (true) o no (false).
- **defaultChecked:** valor booleano que indica el elemento radio debe estar seleccionado por defecto o no
- **length:** número de botones de opción definidos en el grupo de elementos radio.
- **name:** cadena que contiene el valor del parámetro NAME.
- **value:** cadena que contiene el valor del parámetro VALUE.

Y ahora los métodos del objeto Ratio:

- **click():** reproduce la acción que el usuario realiza cuando hace clic en un botón.

Veamos este objeto en un ejemplo. Partiendo del formulario anterior, añadimos un nuevo botón que visualizará en una nueva ventana el valor del botón de radio seleccionado y las casillas seleccionadas. Añadiremos la función `VerSeleccion()`:

```
function VerSeleccion() {  
    var i;  
    var ventana;  
    ventana=window.open("", "", 'width= 340,height=150');  
    ventana.document.write("<B>BOTON SELECCIONADO:</B><BR>");  
    // bucle que recorre los botones  
    for (i=0; i< document.FORMULARIO.BOTONES.length; i++)  
        if (document.FORMULARIO.BOTONES[i].checked) {  
            ventana.document.write("Valor del botón => "+  
                document.FORMULARIO.BOTONES[i].value +"  
                <BR>");  
        }  
    ventana.document.write("<B>CASILLAS SELECCIONADAS:</B><BR>");  
}
```

```

for (i=4; i<7; i++) //casillas elementos 4 a 6
if (document.FORMULARIO.elements[i].checked) {
ventana.document.write("Nombre de casilla: "+
document.FORMULARIO.elements[i].name );
ventana.document.write(" * Valor: " +
document.FORMULARIO.elements[i].value +"<BR>");}
}

```

Y para ver los Ratios

```

<p>
<b>Forma de pago:</b>
<input type="radio" name="BOTONES" value="MENSUAL" checked/>Mensual
<input type="radio" name="BOTONES" value="TRIMESTRAL"/>Trimestral
<input type="radio" name="BOTONES" value="ANUAL"/>Anual
</p>

```

Y el botón que nos abrirá una nueva ventana con las selecciones que hayamos hecho:

```

<input type="button" value="Ver selecciones" name="VER" onclick=
"VerSeleccion () "/>

```

Si ejecutamos veremos la siguiente salida:

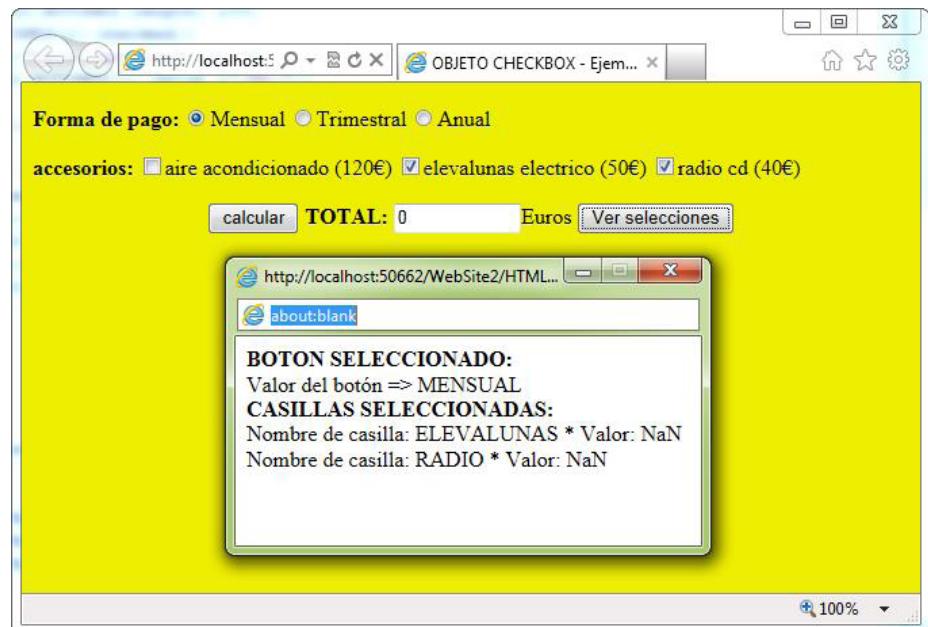


Figura 1.55. Vista del ejemplo1 del objeto Ratio.

Objeto Select

Este objeto representa una lista de opciones dentro de un formulario. Pueden aparecer en una página como cuadros de lista desplegables y por otra como cuadros de lista. Presentan un uso eficaz a la hora de presentar lista de opciones por tratarse de una lista desplegable de la que podremos escoger alguna (o algunas) de sus opciones.

La siguiente figura muestra las listas desplegables y los cuadros de lista:

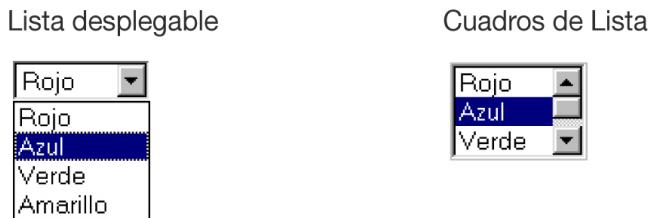


Figura 1.56. Vista de desplegables y cuadros de lista.

Las propiedades y métodos del objeto se resumen en la siguiente tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
length		
name		
options		
selectedIndex		
options[n].defaultSelected	ninguno	onChange=
options[n].index		
options[n].selected		
options[n].text		
options[n].value		

Figura 1.57. Propiedades y métodos del objeto Select.

Para acceder a las propiedades y métodos:

```
[window.]document.nombreformulario.nombrelista.propiedad  

[window.]document.forms[m].nombrelista.propiedad  

[window.]document.nombreformu.nombrelista.options[n].propiedad  

[window.]document.forms[m].nombrelista.options[n].propiedad
```

Las **propiedades del objeto Select** son las siguientes:

- **length:** número de opciones definidas en la lista.

- **name:** contiene el valor del parámetro NAME.
- **options:** se trata de un array que contiene una entrada por cada una de las opciones de la lista, en el mismo orden en el que aparecen en el código HTML. Este array tiene, a su vez, las siguientes propiedades:
 - **defaultSelected:** indica si la opción debe estar seleccionada por defecto (true, sí; false no).
 - **index:** indica la posición de la opción dentro de la lista.
 - **selected:** indica si la opción está actualmente seleccionada o no (true, sí; false no).
- **selectedIndex:** contiene el valor del índice de la opción actualmente seleccionada

Vamos a ver todo esto con un ejemplo, en el que deberemos seleccionar entre varias opciones para verlas posteriormente en una nueva ventana:

Para el primer plato se crea una lista desplegable cuya definición es la siguiente:

```
<p>
<b>Elige el 1º plato:</b>
<select size="1" name="PRIMERPLATO">
  <option value="" selected="selected">selecciona primer plato</option>
  <option value="migas con chorizo">migas con chorizo</option>
  <option value="borrajas">borrajas</option>
  <option value="col de bruselas">coles de bruselas</option>
  <option value="patatas asadas">patatas asadas</option>
</select>
</p>
```

Si no elegimos ningún elemento del menú se visualizará un aviso indicándolo. Para elegir el segundo plato se crea un cuadro de lista cuya definición es la siguiente:

```
<p>
  &nbsp;<b>Elige el 2º plato:</b>
<select size="3" name="SEGUNDOPLATO" multiple="multiple">
  <option>entreco a la plancha</option>
  <option>cochinillo a la plancha</option>
  <option>huevos fritos con longaniza</option>
  <option selected="selected">patatas asadas</option>
</select>
</p>
```

```
</select>
```

```
</p>
```

Para elegir el postre definimos un grupo de botones de radio:

```
<p>
```

```
    <b>Postre:</b>
    <input type="radio" name="POSTRE" value="FRUTA" checked="checked" />fruta
    <input type="radio" name="POSTRE" value="YOGUR"/>yogur
    <input type="radio" name="POSTRE" value="HELADO"/>helado
    <input type="radio" name="POSTRE" value="OTRO"/>otro

```

```
</p>
```

El botón Ver Menú visualiza en una nueva ventana el menú elegido por el usuario. Al hacer clic en el botón se invoca a la función `Verseleccion()` enviando el parámetro `this.form` que hace referencia al formulario actual:

```
<input type="button" name="VER" value="Ver menu"
       onclick="Verseleccion(this.form)"/>
```

El código JavaScript de la función `Verseleccion()` es el siguiente:

```
<script type="text/javascript">
    function Verseleccion(form) {
        var ventana;
        ventana=window.open("", "", 'width=340,height=150');
        ventana.document.write("<B>HAS ELEGIDO EL SIGUIENTE
MENÚ:</B><BR>");
        ventana.document.write("<B>Primer plato:</B>");
        //obtener selección primer plato
        primero = form.PRIMERPLATO.selectedIndex;
        if(form.PRIMERPLATO.options[primero].value=="") {
            alert("NO HAS ELEGIDO PRIMER PLATO");
        }
        //si no elegimos nada
    }
    ventana.document.write(form.PRIMERPLATO
options[primero].value + "<BR>");
    //visualizamos el valor de la opción elegida

```

```

ventana.document.write("<B>Segundo plato:</B>");

segundo = formSEGUNDOPLATO.selectedIndex;

//obtener selección 2º plato

ventana.document.write(formSEGUNDOPLATO.
options[segundo].text+"<BR>");

ventana.document.write("<B>Postre:</B>");

for (i=0; i<formPOSTRE.length; i++)

if (formPOSTRE[i].checked)

ventana.document.write(formPOSTRE[i].value+"<BR>");

ventana.document.close();

//cerramos la escritura en el documento

}

</script>

```

Para obtener el elemento seleccionado de cada una de las listas se han usado estas expresiones:

```

primero = formPRIMERPLATO.selectedIndex;

segundo = formSEGUNDOPLATO.selectedIndex;

```

Para obtener el valor (debe estar definido el parámetro VALUE en la definición de la lista) seleccionado de la primera lista usamos esta expresión:

```
formPRIMERPLATO.options [primero].value
```

Para obtener el texto de la opción seleccionada de la segunda lista usamos esta expresión:

```
formSEGUNDOPLATO.options [segundo].text
```

Si ejecutamos nos dará la siguiente salida:

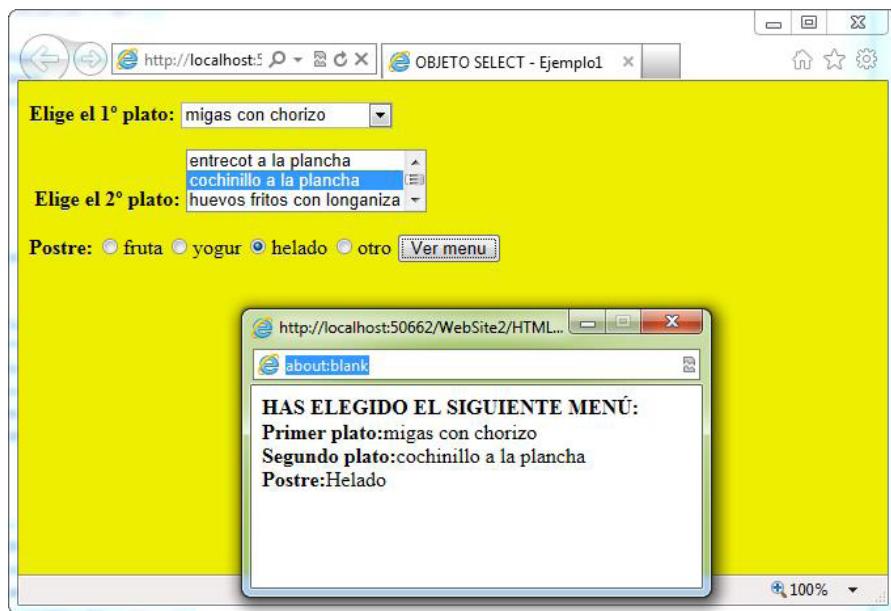


Figura 1.58. Vista del ejemplo1 del objeto Select.

Objeto Hidden

Este es el objeto oculto del formulario, contiene cadenas de caracteres cuyos contenidos no son visibles para el usuario de la página Web. Es similar a un objeto Text salvo que no tiene valor por defecto y que no se puede editar. Son útiles para las aplicaciones que implican múltiples pantallas y se suele utilizar para conservar información de estado entre las páginas.

Las propiedades del objeto Hidden son las siguientes:

- **name:** cadena que contiene el valor del parámetro NAME.
- **value:** cadena que contiene el valor del parámetro VALUE.

Para acceder a las propiedades de este objeto usaremos la siguiente sintaxis:

```
[window.]document.nombreformulario.nombrecampo.propiedad
[window.]document.nombreformulario.elements[n].propiedad
[window.]document.forms[m].nombrecampo.propiedad
[window.]document.forms[m].elements[n].propiedad
```

1.10.4. Propiedades y métodos de los objetos del lenguaje

Los objetos vistos hasta el momento son los que forman parte de la jerarquía de objetos *JavaScript*. Sin embargo *JavaScript* posee otro tipo de objetos propios del lenguaje que permiten manejar nuevas estructuras de datos y añadir utilidades al lenguaje.

Objeto String

Este objeto nos permite la manipulación de cadenas de texto. Cuando asignamos una cadena a una variable, *JavaScript* crea un objeto de tipo *String* que es el que nos permite hacer las manipulaciones.

Las propiedades y métodos del objeto *String* se resumen en esta tabla:

PROPIEDADES	MÉTODOS	MANEJADORES EVENTOS
length prototype	anchor() big() blink() bold() charAt() concat() fixed() fontcolor() fontsize() indexOf() italics() lastIndexOf() link() small() split() strike() slice() sub() substr() substring() sup() toLowerCase() toUpperCase()	

Figura 1.59. Propiedades y métodos del objeto *String*.

Para acceder a las propiedades y métodos:

ObjetoString.Propiedad

ObjetoString.Método()

Las **propiedades del objeto String** son las siguientes:

- **length:** nos indica la longitud en caracteres de la cadena dada.

Ejemplo: "Hola Mundo".length → devuelve 10

- **prototype:** nos permite añadir nuevas propiedades al objeto String.

Los métodos del **objeto String** son los siguientes:

- **anchor(nombre_del_ancla):** crea un enlace (local) asignando al atributo NAME el valor de 'nombre_del_ancla'. Este nombre debe estar entre comillas " ". Se asigna como texto del enlace el que tenga el objeto *String*. Genera el código HTML:

```
<A NAME="nombre_del_ancla">valor del objeto string</A>
```

- **big():** devuelve el valor del objeto string con una fuente grande. Genera el código HTML:

```
<big>valor del objeto string</big>
```

- **blink()**: devuelve el valor del objeto String con un efecto intermitente. Solo funciona actualmente en Mozilla Firefox. Genera el código HTML:

```
<blink>valor del objeto string</blink>
```

- **bold()**: devuelve el valor del objeto String en negrita. Genera el código HTML:

```
<b>valor del objeto string</b>
```

- **charAt(indice)**: devuelve el carácter situado en la posición especificada por ‘índice’ (el primer carácter ocupa la posición 0).

Ejemplo: “Hola Mundo”.charAt(0); à devuelve H

- **concat(cadena)**: concatena el valor del objeto String con el que se pasa como parámetro. C1.concat(C2) → Devuelve la cadena C1 concatenada con la cadena C2. Es equivalente a C1+C2

Ejemplo: “Hola”.concat(“Mundo”); à devuelve HolaMundo

- **fixed()**: devuelve el valor del objeto string con una fuente con formato monoespacio. Genera el código HTML:

```
<tt>valor del objeto string</tt>
```

- **fontcolor(color)**: cambia el color con el que se muestra la cadena. La variable color debe ser especificada entre comillas: “ ”, o bien siguiendo el estilo de HTML. Genera el código HTML:

```
<font color="color">valor del objeto string</font>
```

- **fontsize(tamaño)**: cambia el tamaño con el que se muestra la cadena. Los tamaños válidos son de 1 (más pequeño) a 7 (más grande).

- **indexOf(cadena_buscada, indice)**: devuelve el lugar de la cadena actual donde se encuentra la primera ocurrencia de ‘cadena_buscada’ a partir de la posición dada por ‘índice’. Este último argumento es opcional y, si se omite, la búsqueda comienza por el primer carácter de la cadena. Si no lo encuentra devuelve -1.

Ejemplo: “Hola Mundo”.indexOf(“n”); à devuelve 7

- **italics()**: devuelve la cadena en cursiva. Genera el código HTML:

```
<i>valor del objeto string</i>
```

- **lastIndexOf(cadena_buscada ,indice)**: devuelve el lugar donde se encuentra la última ocurrencia de ‘cadena_buscada’ dentro de la cadena actual, a partir de la posición dada por ‘índice’, y buscando hacia atrás. Este último argumento es opcional y, si se omite, la búsqueda comienza por el último carácter de la cadena.

- ***link(URL)***: crea un enlace donde el atributo HREF toma el valor del URL y se asigna como texto del enlace el que tenga el objeto string. Genera el código HTML:

```
<a href="URL">valor del objeto string</a>
```

- ***small()***: devuelve el valor de la cadena con una fuente pequeña. Genera el código HTML:

```
<small>valor del objeto string</small>
```

- ***split(carácter)***: divide una cadena en subcadenas creadas a partir de la original de la siguiente forma:

- **1º subcadena**: desde el comienzo hasta el carácter especificado (o hasta el final si no lo encuentra)
- **2ª y sucesivas**: a partir del carácter especificado hasta la siguiente ocurrencia del mismo o hasta el final.
- El carácter no se incluye en las subcadenas.

Ejemplo. "Hola Mundo".split(" "); à devuelve "Hola,Mundo"

- ***strike()***: devuelve el valor de la cadena de caracteres tachada. Genera el código HTML:

```
<strike>valor del objeto string</strike>
```

- ***slice(inicio,fin)***: devuelve una cadena formada por los caracteres que se encuentra entre la posición inicio y fin-1.

- ***sub()***: devuelve el valor de la cadena con formato de subíndice. Genera el código HTML:

```
<sub>valor del objeto string</sub>
```

- ***substr(N1,N2)***: por ejemplo si considero C.substr(N1, N2): devuelve una subcadena a partir de la cadena C tomando N2 caracteres desde la posición N1. Si no se especifica, N2 devolverá desde la posición N1 hasta el final de la cadena.

Ejemplo: "Hola Mundo".substr(2,5); à devuelve "Mu".

- ***substring(N1,N2)***: C.substring(N1,N2): devuelve también una cadena a partir de la cadena C, pero en este caso N1 y N2 indican las posiciones de comienzo y de final de la subcadena.

Ejemplo: "Hola Mundo".substring(2,6); à devuelve la "M".

- ***sup()***: devuelve el valor de la cadena con formato de superíndice. Genera el código HTML:

```
<sup>valor del objeto string</sup>
```

- ***toLowerCase()***: devuelve el valor de la cadena en minúsculas.

Ejemplo: "Hola Mundo".toLowerCase(); à devuelve "hola mundo".

- ***toUpperCase()***: devuelve la cadena en mayúsculas.

Ejemplo: "Hola Mundo".toUpperCase(); à devuelve "HOLA MUNDO"

Veamos ahora un ejemplo con las propiedades y métodos descritos anteriormente:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>OBJETO STRING - Ejemplo1</title>
<script type="text/javascript">

    function cadenas() {
        var cad ="Hola Mundo";
        with(document) {
            write("la cadena es:"+cad+"<BR>");
            write("longitud de la cadena:"+cad.length+"<BR>");
            write("metodo anchor:"+cad.anchor("Ancla")+"<BR>");
            write("negrita-bold:"+cad.bold()+"<BR>");
            write("quinto caracter es:"+cad.charAt(5)+"<BR>");
            write("formato FIXED:"+cad.fixed()+"<BR>");
            write("de color rojo - fontcolor:"+cad.fontcolor("#FF0000")+"<BR>");
            write("tamaño 5 - fontsize:"+cad.fontSize(5)+"<BR>");
            write("<BR>en cursiva-Italics:"+cad.italics()+"<BR>");
            write("la primera <b>o</b> esta empezando por detras,");
            write("en la posicion-lastIndexOf:"+cad.lastIndexOf("o")+"<BR>");
            write("haciendola enlace-link:"+cad.link("http://seas.es")+"<BR>");
            write("tachada-strike:"+cad.strike()+"<BR>");
            write("Subindice-sub:"+cad.sub)+"<BR>";
            write("Superindice-sup:"+cad.sup)+"<BR>";
        }
    }
</script>
</head>
<body bgcolor="#eeee00">
<script type="text/javascript">
cadenas();
</script>
```

```

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

```



NOTA

La orden `with` informa a JavaScript que el siguiente grupo de órdenes, dentro de los símbolos de llave, serán referidas a un objeto en particular.

```

with (objeto) {
  [instrucciones]
}

```

La salida generada será la siguiente:



Figura 1.60. Vista del ejemplo1 del objeto String.

Veamos otro ejemplo que muestra en el área el código HTML que se genera al aplicar algunos de los métodos del objeto `String`. El código será el siguiente:

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>OBJETO STRING - Ejemplo2</title>
  <script type="text/javascript">
    function Formateatexto() {
      var texto,colortexto;
      texto=document.forms[0].cadena.value;
      //comprobamos si no se ha tecleado nada
    }
  </script>
</head>
<body>
  <form>
    <input type="text" name="cadena" value="Hola Mundo" />
    <input type="button" value="Formatear" onclick="Formateatexto()" />
  </form>
</body>

```

```

if(document.forms[0].cadena.value=="") {
    alert("¡Has de introducir datos en la cadena!");
    document.forms[0].cadena.focus();
    return;
}

//mayuscula o minuscula

if(document.FORMULARIO.BOTONES[0].checked) texto=texto.toUpperCase();
if(document.FORMULARIO.BOTONES[1].checked) texto=texto.toLowerCase();

//casillas de seleccion

if(document.FORMULARIO.NEGRITA.checked) texto=texto.bold();
if(document.FORMULARIO.ITALICA.checked) texto=texto.italics();
if(document.FORMULARIO.PEQUE.checked) texto=texto.small();
if(document.FORMULARIO.GRANDE.checked) texto=texto.big();

//lista de colores

ct=document.FORMULARIO.COLOR.selectedIndex;
colortexto=document.FORMULARIO.COLOR.options[ct].value;
if(ct>0)
    texto=texto.fontcolor(colortexto);

//añadimos el link

document.FORMULARIO.resul.value=texto.link("http://estudioabiertos.es");
}

</script>
</head>
<body bgcolor="#FFFF99">
<form name="FORMULARIO">
<p>
    <b>Escribe una cadena :</b>
    <input type="text" name="cadena" size="20"/>
    <input type="radio" value="MAYUS" checked="checked" name="BOTONES"/><b>Mayúsculas</b>
    <input type="radio" name="BOTONES" value="MINUS"/><b>Minúsculas</b>

```

```

</p>
<p>
<b>
    <input type="checkbox" name="NEGRITA"/>Negrita
    <input type="checkbox" name="ITALICA"/>Itálica
    <input type="checkbox" name="PEQUE"/>Pequeña
    <input type="checkbox" name="GRANDE"/>Grande</b>
<select size="1" name="COLOR">
    <option selected="selected" value="">Elige un color</option>
    <option value="#ff0000">Rojo</option>
    <option value="#0000ff">Azul</option>
    <option value="#ffff00">Amarillo</option>
    <option value="#00ff00">Verde</option>
</select>
<textarea rows="5" name="resul" cols="28"
readonly="readonly" wrap>
    Cadena resultante:</textarea></p>
<p>
<center>
    <input type="button" onclick="Formatateatexto()"
        value="Ver cadena resultante en forma de URL"
        name="boton"/>
</center>
</p>
</form>
<hr/>
</body>
</html>

```

La salida será la siguiente:

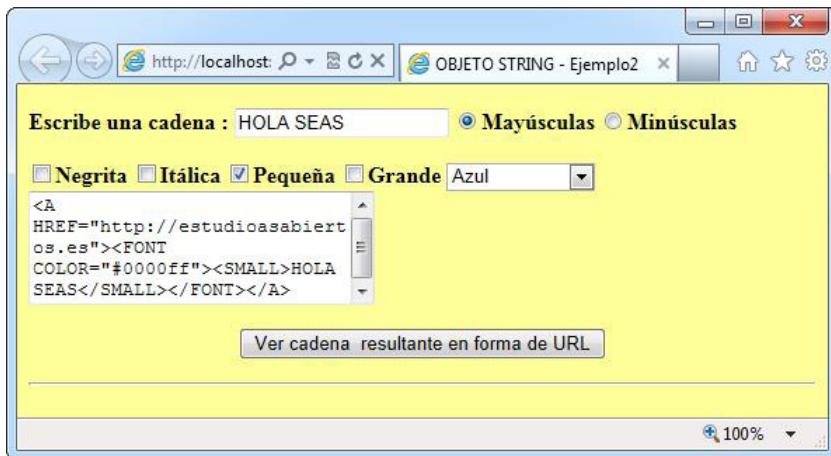


Figura 1.61. Vista del ejemplo2 del objeto String.

Objeto Math

Este objeto nos permite poder realizar cálculos en los scripts. Sus propiedades sólo pueden consultarse, son constantes matemáticas de uso frecuente. Las referencias a las propiedades devuelven los valores inherentes (como por ejemplo pi π); las referencias a los métodos requieren un valor enviado como parámetro al método y devuelven el resultado del método realizando la operación con el parámetro. La forma de usar el objeto *Math* es la misma que la forma de usar cualquier objeto JavaScript:

`Math.propiedad`

`Math.método ()`

Las propiedades del objeto *Math* son las siguientes:

PROPIEDAD	DESCRIPCIÓN
<code>Math.E</code>	Constante de Euler. Número “e”, base de los logaritmos naturales (neperianos) (aproximadamente 2.718).
<code>Math.LN2</code>	Logaritmo neperiano de 2 (aproximadamente 0.693).
<code>Math.LN10</code>	Logaritmo neperiano de 10 (aproximadamente 2.302).
<code>Math.LOG2E</code>	Logaritmo en base 2 de “e” (aproximadamente 1.442).
<code>Math.LOG10E</code>	Logaritmo en base 10 de “e” (aproximadamente 0.434).
<code>Math.PI</code>	Constante PI (aproximadamente 3.14159).
<code>Math.SQRT1_2</code>	Raíz cuadrada de 0.5 (aproximadamente 0.707).
<code>Math.SQRT2</code>	Raíz cuadrada de 2 (aproximadamente 1.414).

Figura 1.62. Propiedades del objeto *Math*.

Veamos ahora algunos **métodos del objeto Math** con ejemplos:

FUNCIÓN	MÉTODO	EXPLICACIÓN Y EJEMPLO
Raíz cuadrada	Math.sqrt(N)	Devuelve la raíz cuadrada del argumento. Ej: Math.sqrt(27) devuelve 5.196152422706632
Potencia	Math.pow(N1, N2)	Devuelve N1 elevado a N2. Ej: Math.pow(5, 3) devuelve 125
Valor absoluto	Math.abs(N)	Devuelve el valor absoluto de un número (es decir, el valor de ese número sin signo). Ej: Math.abs(-5.3) devuelve 5
Redondeo	Math.round(N)	Devuelve el número entero más próximo al número N. Ej: Math.round(27.2) devuelve 27 Math.round(27.5) devuelve 28 Math.round(-27.5) devuelve -27 Math.round(27.6) devuelve 28
Entero superior	Math.ceil(N)	Devuelve el entero más cercano (por arriba) al número N. Si N es un número entero devuelve N. Ej: Math.ceil(6.1) devuelve 7 Math.ceil(-6.1) devuelve -6
Entero inferior	Math.floor(N)	Redondea el número al valor entero inmediatamente inferior. Si N es un número entero devuelve N. Ej: Math.floor(6.1) devuelve 6 Math.floor(-6.1) devuelve -7 Math.floor(6.7) devuelve 6 Math.floor(-6.7) devuelve -7
Número aleatorio	Math.random()	Genera un número aleatorio entre 0 y 1. A continuación podemos ver el valor devuelto por Math.random() en tres ejecuciones consecutivas. 0.46879380653891483 0.26926274793632804 0.3076617575535776
Máximo	Math.max(N1, N2)	Devuelve el número mayor de los dos números que se pasan como argumento. Ej: Math.max(2, 4) devuelve 4
Mínimo.	Math.min(N1, N2)	Devuelve el número menor de los dos números que se pasan como argumento. Ej: Math.min(2, 4) devuelve 2

Figura 1.63. Métodos y ejemplos del objeto Math.

Aquí tenemos otros métodos del objeto Math:

MÉTODO	DEVUELVE
Math.acos(N)	Función arcocoseno. Devuelve el arcocoseno de N en radianes.
Math.asin(N)	Función arcoseno. Devuelve el arcoseno de N en radianes.
Math.atan(N)	Función arcotangente. Devuelve el arcotangente de N en radianes.
Math.cos(N)	Devuelve el coseno de N.
Math.exp(N)	Devuelve el valor e^{numero} .
Math.floor(N)	Devuelve el siguiente número entero menor o igual que N.
Math.log(N)	Devuelve el logaritmo neperiano de N.
Math.sin(N)	Devuelve el seno de N en radianes.
Math.tan(N)	Devuelve la tangente de N en radianes.

Figura 1.64. Otros métodos del objeto Math.

En todas las órdenes que interviene el objeto *Math* se puede utilizar referencia abreviada. Por ejemplo, la siguiente expresión:

```
Resultado = Math.sqrt(125) * Math.PI
```

Se puede sustituir por esta otra:

```
with(Math) {
    Resultado = sqrt(125) * PI;
}
```

Vamos a aplicar en el siguiente ejemplo el uso de algunos métodos del objeto *Math*. Se escribirá una cantidad numérica y al hacer clic en el botón “Obtener cálculos” se visualizarán los cálculos obtenidos al aplicar algunos métodos. El código será el siguiente:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>OBJETO MATH - Ejemplo1</title>
    <script type="text/javascript">
        function calculosmatematicos(numero) {
            with(Math) {
                document.forms[0].RAIZ.value=sqrt(numero);
                document.forms[0].SUPERIOR.value=ceil(numero);
                document.forms[0].ELEVADO.value=pow(numero,3);
                document.forms[0].INFERIOR.value=floor(numero);
                document.forms[0].ABSOLUTO.value=abs(numero);
            }
        }
    </script>
</head>
<body>
    <form>
        <input type="button" value="Obtener cálculos" onclick="calculosmatematicos(document.forms[0].NUMERO.value)">
        <input type="text" name="RAIZ" value="">
        <input type="text" name="SUPERIOR" value="">
        <input type="text" name="ELEVADO" value="">
        <input type="text" name="INFERIOR" value="">
        <input type="text" name="ABSOLUTO" value="">
    </form>
</body>

```

```

        document.forms[0].REDONDEO.value=round(numero);
        document.forms[0].NEPERIANO.value=log(numero);
        document.forms[0].EXPONENTE.value=exp(numero);
    }
}

</script>
</head>
<body bgcolor="#FFFF99">
<form name="FORMULARIO">
<center>
<p>
<b>Escribe una cantidad numerica :</b>
<input type="text" name="numero" size="20"/>
</p>
<p>
<b>raiz cuadrada :</b>
<input type="text" name="RAIZ" size="20" readonly="readonly"/>
</p>
<p>
<b>elevado a 3:</b>
<input type="text" name="ELEVADO" size="20" readonly="readonly"/>
</p>
<p>
<b>valor absoluto :</b>
<input type="text" name="ABSOLUTO" size="20" readonly="readonly"/>
</p>
<p>
<b>logaritmo neperiano :</b>
<input type="text" name="NEPERIANO" size="20" readonly="readonly"/>
</p>
<p>
<b>redondeo superior :</b>
<input type="text" name="SUPERIOR" readonly="readonly"/>
</p>
<p>
<b>entero inferior :</b>
<input type="text" name="INFERIOR" size="20" readonly="readonly"/>
</p>

```

```

<p>
<b>redondeo :</b>
<input type="text" name="REDONDEO" size="20" readonly="readonly"/>
</p>
<p>
<b>exponente :</b>
<input type="text" name="EXPONENTE" readonly="readonly"/>
</p>
<p>
<input type="button" value="Obtener calculos"
       name="boton" onclick="calculosmatematicos(document.
forms[0].numero.value)">
</p>
</center>
</form>
<hr>
</body>
</html>

```

Si ejecutamos, obtendremos la siguiente salida cuando pulsamos en “Obtener cálculos”:

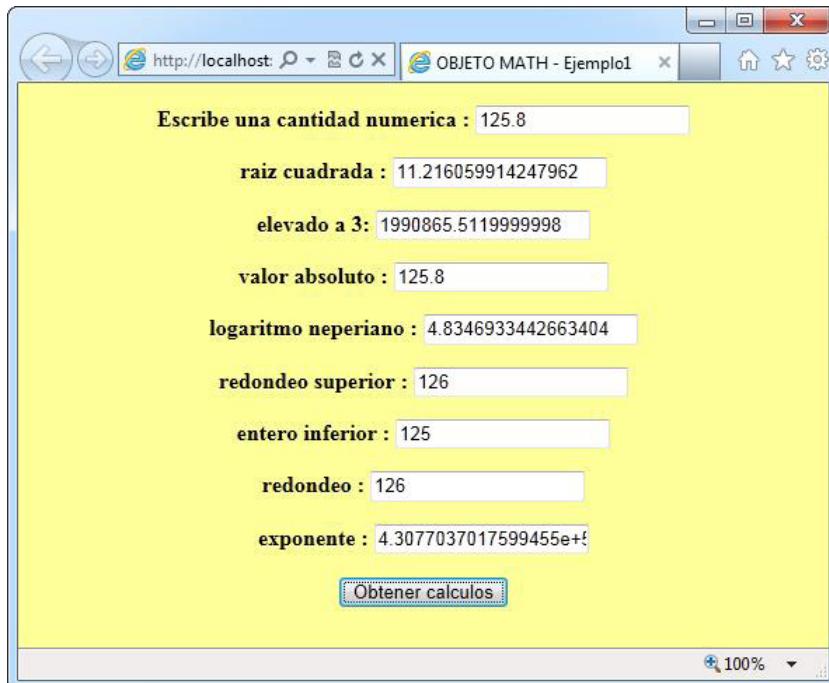


Figura 1.65. Vista del ejemplo1 del objeto Math.

Objeto Date

Este objeto nos va a permitir manipular fechas y horas. Dispone de diversos métodos para obtener y modificar el año, el mes, el día, las horas, los minutos y los segundos. Debemos tener en cuenta lo siguiente:

- *JavaScript* maneja fechas en milisegundos.
- Los meses de enero a diciembre vienen dados por un entero cuyo rango varía entre el 0 y el 11 (el mes 0 es Enero, el mes 1 es Febrero, y así sucesivamente).
- Los días de la semana de domingo a sábado vienen dados por un entero cuyo rango varía entre 0 y 6 (el día 0 es el domingo, el día 1 es el lunes, ...).
- Los años se ponen tal cual, y las horas se especifican con el formato HH:MM:SS.

La sintaxis básica para **crear un objeto Fecha** es la siguiente:

```
ObjetoFecha = new Date([parámetros])
```

Podemos crear un objeto Date vacío (sin parámetros), entonces se creará con la fecha correspondiente al momento actual en el que se crea.

Ejemplo:

```
var Fecha=new Date();
```

O podemos crearlo dándole una fecha concreta (con parámetros). Indicando parámetros tenemos estas posibilidades:

```
var Fecha = new Date("Month dd, yyyy hh:mm:ss");
var Fecha = new Date("Month dd, yyyy");
var Fecha = new Date(yyyy,mm,dd, hh, mm, ss);
var Fecha = new Date(yyyy,mm,dd);
var Fecha = new Date(milisegundos);
```

Donde:

- Month → nombre de mes.
- dd → día.
- yyyy → año.
- hh → horas.
- m → minutos.
- ss → segundos.

El nombre del mes debe aparecer completo y en inglés. Las horas, minutos y segundos han de ir separados por dos puntos (:)

Por ejemplo, podemos almacenar el 27 de julio de 2001 de varias formas:

```
var Fecha1 = new Date(2001, 6, 27);

var Fecha2 = new Date("July 27, 2001");
```

Para utilizar los métodos del objeto Date:

```
ObjetoFecha.método()
```

Veamos los métodos del objeto Date:

MÉTODO	DESCRIPCIÓN
ObjetoFecha.getTime()	Devuelve el número de milisegundos transcurridos desde el 1 de enero de 1970 hasta el momento actual.
ObjetoFecha.getDate()	Devuelve el día del mes actual como un valor entero entre 1 y 31.
ObjetoFecha.getDay()	Devuelve el día de la semana actual como valor un entero entre 0 y 6.
ObjetoFecha.getHours()	Devuelve la hora del día actual como un valor entero entre 0 y 23.
ObjetoFecha.getMinutes()	Devuelve los minutos de la hora actual como un valor entero entre 0 y 59.
ObjetoFecha.getMonth()	Devuelve el mes del año actual como un valor entero entre 0 y 11 (enero=0).
ObjetoFecha.getSeconds()	Devuelve el número de segundos de la hora actual como un valor entero entre 0 y 59.
ObjetoFecha.getYear()	Devuelve el año actual como un valor entero. Si el año se encuentra entre 1900 y 1999, devuelve un entero de dos dígitos (diferencia entre el año y 1900). Si está fuera de este rango, Explorer devuelve el valor del año expresado en 4 dígitos y Netscape devuelve el año expresado como la diferencia entre este y 1900.
ObjetoFecha.setDate(día_mes)	Establece el día del mes en el objeto Date (valor entre 1 y 31)
ObjetoFecha.setDay(día_semana)	Establece el día de la semana (valor entre 0 y 6, domingo=0).
ObjetoFecha.setHours(horas)	Establece la hora del objeto Date (valor entre 0 y 23).
ObjetoFecha.setMinutes(minutos)	Establece los minutos del objeto Date (valor entre 0 y 59).
ObjetoFecha.setMonth(mes)	Establece el mes del objeto Date (valor entre 0 y 11).
ObjetoFecha.setSeconds(segundos)	Establece el número de segundos del objeto Date (valor entre 0 y 59).
ObjetoFecha.setTime(milisegundos)	Establece el número de milisegundos transcurridos desde el 1 de enero de 1970 y a las 00:00:00 horas.
ObjetoFecha.setYear(año)	Establece el año del objeto Date. Si se indica un número entre 0 y 99, este método asigna como año ese valor más 1900. Si el año indicado está fuera de ese rango, el método asigna el valor tal cual.
ObjetoFecha.toGMTString()	Devuelve la fecha en forma de cadena usando la convención de zona horaria del meridiano de Greenwich (GMT).

MÉTODO	DESCRIPCIÓN
ObjetoFecha.toLocaleString()	Convierte la fecha del objeto Date en una cadena en el formato del sistema.

Figura 1.66. Metodos y descripción del objeto Date.

Veamos el uso de alguno de los métodos del objeto Date en el siguiente ejemplo:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>OBJETO DATE - Ejemplo1</title>
    <script type="text/javascript">

        function Fechas (F) {
            document.write("Fecha =>" + F + "<br>");

            document.write("dia del mes - getDate(): =>" + F.
                getDate() + "<br>");

            document.write("dia de la semana - getDay(): =>" + F.
                getDay() + "<br>");

            document.write("numero de mes - getMonth(): =>" + F.
                getMonth() + "<br>");

            document.write("año - getYear(): =>" + F.
                getYear() + "<br>");

            document.write("horas:minutos:segundos:
=>" + F.getHours() + ":" + F.getMinutes() + ":" + F.
                getSeconds() + "<br>");

            document.write("toGMTString() =>" + F.
                toGMTString() + "<br>");

            document.write("toLocaleString() =>" + F.
                toLocaleString() + "<br><hr>");

        }

    </script>
</head>
<body bgcolor="#FFFF99">
    <script type="text/javascript">
        var Fecha1 = new Date("July 27, 2001");
        var Fecha2 = new Date();
        Fechas(Fecha1);
        Fechas(Fecha2);
    </script>
```

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

La salida generada utilizando el navegador Internet Explorer es:

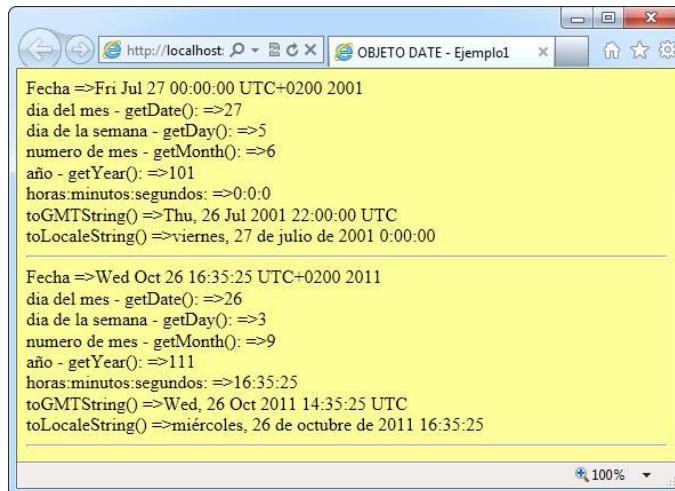


Figura 1.67. Vista del ejemplo1 del objeto Date en Internet Explorer.

Con el navegador Mozilla Firefox:

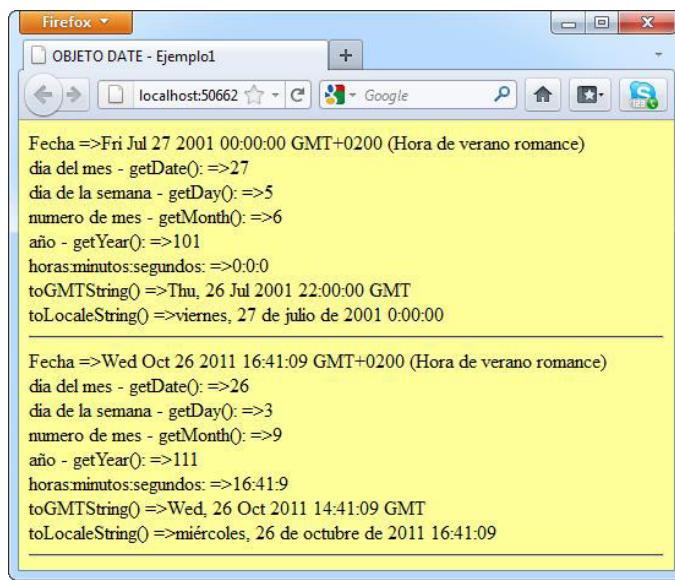


Figura 1.68. Vista del ejemplo1 del objeto Date en Mozilla Firefox.

Y con Google Chrome:

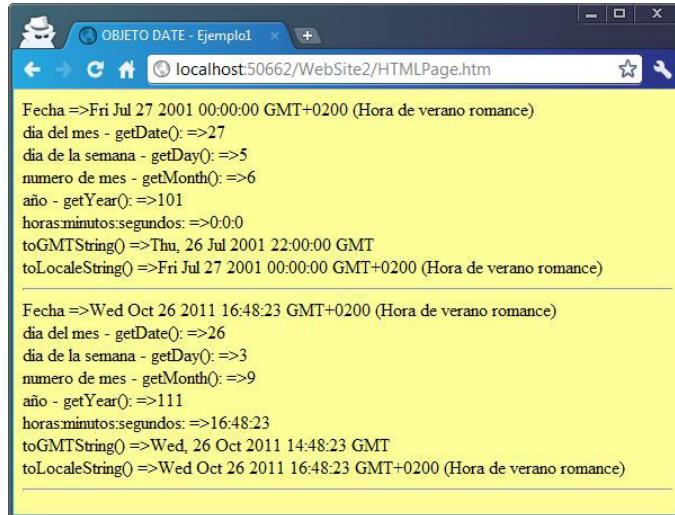


Figura 1.69. Vista del ejemplo1 del objeto Date en Google Chrome.

Veamos otro ejemplo donde definimos una función que recibe una fecha y la devuelve personalizada, incluyendo el nombre del día de la semana y el nombre del mes. El código del script es el siguiente:

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>OBJETO DATE - Ejemplo2</title>
<script type="text/javascript">
meses=new Array();
dias=new Array();
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";
dias[1]="lunes";        dias[2]="martes";
dias[3]="miercoles";    dias[4]="jueves";
dias[5]="viernes";      dias[6]="sabado";
dias[0]="domingo";

```

```

function Convertir(F) {
    dia=dias[F.getDay()];
    mes=meses[F.getMonth()];
    an=F.getFullYear();
    if(navigator.appName=="Netscape" || navigator.appName.
    indexOf("Explorer") != -1){
        an=an+1900;
    }
    return dia +", "+F.getDate()+" de "+mes+" de "+an;
    document.write(Convertir(new Date()));
}

</script>
</head>
<body bgcolor="#FFFF99">
</body>
</html>

```

Con el código:

```

if(navigator.appName=="Netscape" ||
navigator.appName.indexOf("Explorer") != -1)
{
    an=an+1900;
}

```

Detectamos si el navegador con el que se está navegando es Internet Explorer o Mozilla Firefox, y le sumamos 1900, por lo que se explica del método `getFullYear()` en la tabla de los métodos del objeto Date. Si ejecutamos veremos lo siguiente:

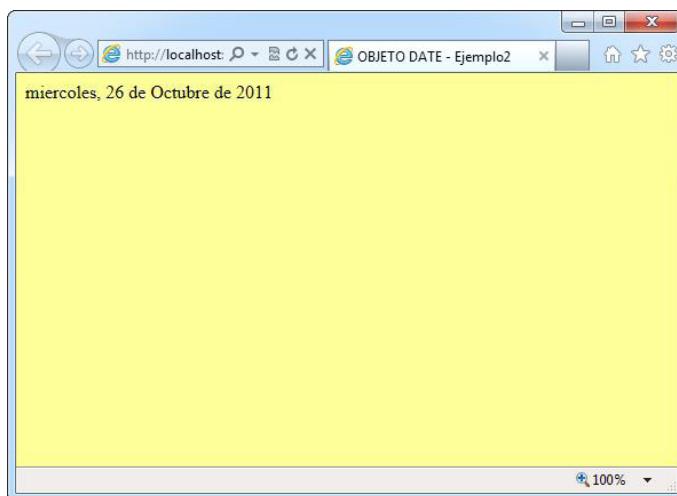


Figura 1.70. Vista del ejemplo2 del objeto Date.

RESUMEN

- Hemos visto como surge JavaScript, como ha ido evolucionando a lo largo de los años y como se ha ido adaptando a los diversos y nuevos navegadores que han ido surgiendo, evolucionando con estos y adaptándose a las nuevas tecnologías que han ido surgiendo. También hemos aprendido su sintaxis, como incluirlo en las páginas XHTML, sus limitaciones y posibilidades y los diferentes incompatibilidades con los navegadores.
- Hemos aprendido las variables y sus tipos, como declararlas y como usarlas. También hemos descubierto los operadores para poder implementar funcionalidades más avanzadas.
- Hemos estudiado y practicado con las estructuras de control de flujo, con las que realizar cosas más complejas en nuestros scripts, pudiendo realizar tomas de decisiones y scripts. Hemos visto la estructura If...else, la estructura For, While, Do...While, For...In y la utilización de Arrays mediante estructuras de control.
- Despues hemos visto también las funciones, que nos permiten ejecutar una serie de instrucciones cuando se produce un evento, devolviéndonos o no un valor, y pudiéndolas ejecutar tantas veces como se deseé, con solo llamarla con el nombre asignado. Hemos aprendido a crearlas, donde declararlas, como llamarlas, como asignarles y pasársele los argumentos. También hemos visto las funciones ya predefinidas en JavaScript.
- Hemos conocidos los objetos, su jerarquía y su referencia:
 - Hemos visto los Objetos del Navegador, sus objetos predefinidos, sus propiedades y sus métodos.
 - Hemos explicado también los Objetos del Documento, sus objetos predefinidos, sus propiedades y sus métodos.
 - Por último, hemos presentado los Objetos del Lenguaje, sus objetos predefinidos, sus propiedades y sus métodos.

