---

1. INTRODUCCIÓN

1.1. ¿QUÉ ES JAVASCRIPT?

JavaScript® es el lenguaje interpretado orientado a objetos desarrollado por Netscape que se utiliza en millones de páginas web y aplicaciones de servidor en todo el mundo. JavaScript de Netscape es un superconjunto del lenguaje de scripts estándar de la edición de ECMA-262 3 (ECMAScript) que presenta sólo leves diferencias respecto a la norma publicada.

Contrariamente a la falsa idea popular, JavaScript no es "Java interpretativo". En pocas palabras, JavaScript es un lenguaje de programación dinámico que soporta construcción de objetos basado en prototipos. La sintaxis básica es similar a Java y C++ con la intención de reducir el número de nuevos conceptos necesarios para aprender el lenguaje. Las construcciones del lenguaje, tales como sentencias if, y bucles for y while, y bloques switch y try ... catch funcionan de la misma manera que en estos lenguajes (o casi).

JavaScript puede funcionar como lenguaje procedimental y como lenguaje orientado a objetos. Los objetos se crean programáticamente añadiendo métodos y propiedades a lo que de otra forma serían objetos vacíos en tiempo de ejecución, en contraposición a las definiciones sintácticas de clases comunes en los lenguajes compilados como C++ y Java. Una vez se ha construido un objeto, puede usarse como modelo (o prototipo) para crear objetos similares.

Las capacidades dinámicas de JavaScript incluyen construcción de objetos en tiempo de ejecución, listas variables de parámetros, variables que pueden contener funciones, creación de scripts dinámicos (mediante eval), introspección de objetos (mediante for ... in), y recuperación de código fuente (los programas de JavaScript pueden decompilar el cuerpo de funciones a su código fuente original).

Los objectos intrínsecos son Number, String, Boolean, Date, RegExp y Math.

¿Qué implementaciones de JavaScript están disponibles?

Mozilla.org alberga dos implementaciones de JavaScript. La primera implementación de JavaScript fue creada por Brendan Eich en Netscape, y desde entonces ha sido actualizada (en JavaScript 1.5) para cumplir con ECMA-262 Edición 5. Este motor, cuyo nombre en código es SpiderMonkey, se implementa en C. El motor Rhino, creado principalmente por Norris Boyd (también en Netscape) es una implementación de Javascript en Java. Al igual que SpiderMonkey, Rhino cumple con ECMA-262 Edición 3.

Varias optimizaciones tales como TraceMonkey (Firefox 3.5), JägerMonkey (Firefox 4) e IonMonkey fueron agregadas al motor de JavaScript de SpiderMonkey durante el tiempo.

Además de las implementaciones anteriores, existen otros motores de JavaScript populares tales como:

V8 de Google, el cual es usado en el navegador Google Chrome.

JavaScriptCore (SquirrelFish/Nitro) usado en algunos navegadores basados en WebKit tales como Apple Safari.

Carakan en Opera.

El motor Chakra, usado en Internet Explorer, es tecnicamente un motor de JScript, en lugar de un motor de JavaScript.

Cada motor de JavaScript de mozilla.org expone una API pública que las aplicaciones pueden llamar para aprovechar el soporte de JavaScript. Por el momento, el entorno de host más común de JavaScript son los navegadores web porque suelen utilizar la API pública para crear "objetos de host" responsables de reflejar el DOM en JavaScript.

Otra aplicación común de JavaScript es como lenguaje interpretado de lado del servidor (web). Un servidor web escrito en JavaScript podría exponer objetos host que representen objetos de una petición y una respuesta HTTP, los cuales podrían ser manipulados por un programa en JavaScript para generar páginas web de manera dinámica.

1.2. ¿Cómo introducir JavaScript en el código HTML?

Básicamente existen dos formas de introducir un *script* de JavaScript en una página HTML:

- Embebido en el código HTML, entre las *tags* **<script>** y **</script>**. El siguiente código muestra un ejemplo de código JavaScript embebido en el HTML de una

página. Como se observa, el código JavaScript figura entre las marcas de comentario **<!--** y **-->**, se introducen para evitar que se generen errores en navegadores que no soporten JavaScript y para que estos navegadores no muestren el código del *script* en la página.

```
<SCRIPT LANGUAGE="JavaScript">
<!--
Programa JavaScript
//-->
</SCRIPT>
<HEAD><TITLE>Introducción a JavaScript</TITLE>
 <SCRIPT LANGUAGE="JavaScript">
<!--
function saludo() {
window.alert('Bienvenido a JavaScript')
}
//-->
</SCRIPT>
</HEAD>
<BODY onLoad="saludo()">
```

- Como archivo .js que se carga con la página HTML. Para ello, debe indicarse en las *tags* anteriores el nombre y ubicación del archivo .js que contiene el *script* JavaScript, como en este ejemplo:

```
<HTML>
<HEAD>
<TITLE>Tutorial de JavaScript</TITLE>
<SCRIPT LANGUAGE="JavaScript"
SRC="scripts/fuente.js"></SCRIPT>
</HEAD>
```

1.3. PRIMEROS PASOS

Vamos a realizar nuestro primer "programa" en JavaScript.

Este script creará un botón que cuando es presionado muestra una ventana diciendo '**HolaMundo.html**'. ¿Qué está sucediendo en este script? Primero la función

se carga y es guardada en memoria. Entonces un botón es hecho con el tag normal (HTML). Hay algo completamente nuevo con el tag. Allí puede ver 'onclick'. Esto le dice al browser que función tiene que invocar cuando este botón es presionado. Existe otra cosa nueva en este script el método 'alert'. Este método ya es declarado en JavaScript- solo se necesita invocarlo. Existen muchos métodos diferentes los cuales se pueden invocar.

Ejem1.html

```
<HTML>
<HEAD>
 <SCRIPT LANGUAGE="JavaScript">
        function HolaMundo() {
            alert("¡Hola, mundo!");
        }
</SCRIPT>
</HEAD>
<BODY>
<FORM>
  <INPUT TYPE="button" NAME="Boton" VALUE="Pulsame"
       onClick="HolaMundo()">
</FORM>
</BODY></HTML>
```

Ahora vamos a ver, paso por paso, que significa cada uno de los elementos extraños que tiene la página anterior:

```
<SCRIPT LANGUAGE="JavaScript">
</SCRIPT>
```

Dentro de estos elementos será donde se puedan poner funciones en JavaScript. Puedes poner cuantos quieras a lo largo del documento y en el lugar que más te guste. Yo he elegido la cabecera para hacer más legible la parte HTML de la página. Si un navegador no acepta JavaScript no leerá lo que hay entre medias de estos elementos.

```
function HolaMundo() {
```

```
alert("¡Hola, mundo!");
 }
```

Esta es nuestra primera función en JavaScript. Aunque JavaScript esté orientado a objetos no es de ningún modo tan estricto como Java, donde nada está fuera de un objeto. En el código de la función vemos una llamada al método alert (que pertenece al objeto window) que es la que se encarga de mostrar el mensaje en pantalla.

```
<FORM>
  <INPUT TYPE="button" NAME="Boton" VALUE="Pulsame"
      onClick="HolaMundo()">
</FORM>
```

Dentro del elemento que usamos para mostrar un botón vemos una cosa nueva: onClick. Es un controlador de evento. Cuando el usuario pulsa el botón, el evento click se dispara y ejecuta el código que tenga entre comillas el controlador de evento onClick, en este caso la llamada a la función HolaMundo(), que tendremos que haber definido con anterioridad. Existen muchos más eventos que iremos descubriendo según avancemos. En el cuarto capítulo hay un resumen de todos ellos.

En realidad, podríamos haber escrito lo siguiente:

```
<FORM>
  <INPUT TYPE="button" NAME="Boton" VALUE="Pulsame"
     onClick="alert('¡;Hola,Mundo!')">
</FORM>
```

y nos habríamos ahorrado el tener que escribir la función y todo lo que le acompaña. Sin embargo me pareció conveniente hacerlo de esa otra manera para mostrar más elementos del lenguaje en el ejemplo.

Ejem2.html

```
<HTML>
<HEAD><TITLE>Mi primer JavaScript!</TITLE></HEAD>
<BODY>
```

```
<BR>
Este es documento normal en HTML.
<BR>
 <SCRIPT LANGUAGE="JavaScript">
  document.write("Esto es JavaScript!")
 </SCRIPT>
<BR>
Otra vez en HTML.
</BODY>
</HTML>
```

La siguiente propiedad de JavaScript la puede observar moviendo el puntero del mouse sobre este link. Tan solo observe la barra de estado en la parte baja del browser. Esto se puede combinar muy bien con funciones de JavaScript. Si mueve el mouse sobre este link una ventana se abrirá. Ahora le mostraré la fuente que produce estos dos efectos:

Ejem3.html (añadido a ejem2)

```
<a href="tpage.htm" onMouseOver="window.status='Otro link estúpido...'; return true">enlace</a>
```

La única cosa que tiene que hacer es agregar el método onMouseOver en su tag <a>. El window.status le permite escribir cosas en la barra de estado de su browser. Como puede ver, tiene que alternar con las comillas. No puede usar " todo el tiempo, porque de otra manera JavaScript no es capaz de identificar el string que quiere imprimir en la barra de estado. Después del string se tiene que escribir ;return true.

El segundo ejemplo usa JavaScript llamando la función 'alert'. Usa el método 'onMouseOver' y la función hello() es invocada cuando este evento ocurre.

Ejem4.html

```
<html>
<head>
<script language="JavaScript">
<!-- Escondemos la funcion
   function hello() {alert("Hola!");
```

```
    }
// -->
</script>
</head>
<body>
<a href="" onMouseOver="hello()">link</a>
</body></html>
```

1.4. Inclusión de ficheros externos con código JavaScript

Los *scripts* que queramos utilizar en una página suelen escribirse en la misma página, normalmente entre las etiquetas **<head>** y **</head>**. Determinados *scripts* pueden aparecer entre las etiquetas **<body>** y **</body>** (por ejemplo, gestores de eventos, o *scripts* que escriben código *on-line*), pero lo normal es que la mayoría de las funciones estén en la cabecera de la página.

Otra posibilidad es la inclusión del código JavaScript en ficheros externos, de extensión **.js**. Estos ficheros son enlazados desde la página HTML donde se utilizan con este código (que deberá ir también entre **<head>** y **</head>**):

```
<SCRIPT LANGUAGE="JavaScript"
SRC="fichero.js"></SCRIPT>
```

Este código es ignorado automáticamente por los navegadores que no soportan JavaScript.

2. ELEMENTOS BÁSICOS

2.1. COMENTARIOS

Un comentario es una parte de nuestro programa que el ordenador ignora y que, por tanto, no realiza ninguna tarea. Se utilizan generalmente para poner en lenguaje humano lo que estamos haciendo en el lenguaje de programación y así hacer que el código sea más comprensible.

En JavaScript existen dos tipos de comentarios.

- El primero nos permite que el resto de la línea sea un comentario. Para ello se utilizan dos barras inclinadas:

```
// comentario de una línea
```

- Sin embargo, también permite un tipo de comentario que puede tener las líneas que queramos. Estos comentarios comienzan con /* y terminan por */. Por ejemplo:

  /* Aquí comienza nuestro maravilloso comentario Que sigue por aquí e indefinidamente hasta que le indiquemos el final */

2.2. LITERALES

Los literales son datos que comprenden números o caracteres usados para representar varios valores fijos en JavaScript. Son valores que no cambian durante la ejecución del Script.

2.2.1. Literales enteros.

Los enteros se pueden representar en formato decimal (base10), octal (base 8) o hexadecimal (base 16). Los literales enteros en decimal pueden incluir cualquier secuencia de dígitos que no comiencen por 0 (cero). Los literales enteros en formato octal comienzan por un cero y pueden incluir cualquier secuencia de dígitos entre el 0 y el 7. Para designar a un hexadecimal, hay que poner 0X delante del entero.

3434 0526 0XF25A

2.2.2 Literales en Coma Flotante

Los literales en coma flotante representan números decimales con parte fraccionaria. Se pueden expresar en forma estándar o con notación científica. En notación científica se utiliza la letra "E" o "e" para designar el exponente. Ambos, el número decimal y el exponente pueden ser positivos o negativos, como se muestra en los ejemplos siguientes:

3405.673 -1.958 8.3200e+ll 8.3200ell

2.2.3. Literales Booleanos

JavaScript implementa tipos de datos booleanos y además soporta los dos literales, "true" y "false". Estos literales representan los valores "1" y "0" respectivamente. Las palabras reservadas *true* y *false* tienen que aparecer en **minúsculas**.

2.2.4 Literales de Cadena

Una cadena está formada por cero o más caracteres encerrados entre dobles comillas (") o entre comillas simples ('). Siempre se debe utilizar el mismo tipo de comilla para rodear cada cadena. Los siguientes son ejemplos de literales:

"saludos" 'saludos' "#12-6" "¡Sonríe por favor!"

'incluye "dobles" comillas'

Como ocurre en el último ejemplo, en algunos casos se puede entremezclar el uso de los dos tipos de entrecomillado, principalmente para insertar una cadena literal dentro de otra.

2.2.5 Caracteres Especiales

Cuando se escriben scripts, a veces es necesario decirle al ordenador que utilice caracteres especiales o teclas, tales como la tabulación o el retorno de carro.

Para hacer esto, se utiliza el carácter "\" delante de un código de escape, como muestra la siguiente tabla:

Carácter	Significado	Carácter	Significado
\n	Nueva línea	\\	Barra invertida
\t	Tabulador	\b	Espacio hacia atrás
\'	Comilla simple	\r	Retorno de carro
\"	Comilla doble		

2.3. SENTENCIAS Y BLOQUES

Los programas de JavaScript constan de ***sentencias***, todas las sentencias terminan en un **punto y coma (;).** Normalmente, el punto y coma se utiliza al final de una línea, pero se pueden poner muchas sentencias en una sola línea separándolas mediante el punto y coma.

Es bastante frecuente en programación tener la necesidad de agrupar muchas sentencias en un solo conjunto, de forma que se puedan tratar como una sola entidad, eso suele hacerse en las funciones y en los condicionales.

Estos agrupamientos de sentencias en una sola entidad se denominan ***bloque.*** Un bloque se crea rodeando todas las sentencias que contenga entre llaves.

```
{
        document.write("Esta línea es la primera sentencia");
        document.write("Esta línea es la segunda sentencia");
        document.write("Todas estas sentencias forman un bloque");
}
```

3. VARIABLES

Las variables son nombres que ponemos a los lugares donde almacenamos la información. Una variable es el nombre que se le da a una posición de memoria en la cual se almacena información. Conociendo esta posición de memoria (dirección) somos capaces de encontrar, actualizar, o recuperar información cuando la necesitemos durante el resto del programa. A través de las variables se pueden asignar nombres significativos a las posiciones donde se almacena la información.

3.1. NOMBRADO DE VARIABLES

El nombre de las variables en JavaScript está formado por una o más letras, dígitos o guiones bajos (_). **No pueden comenzar por un dígito** ("0 …9"). Las letras incluyen los caracteres en mayúsculas ("A …Z") y en minúsculas ("a…z"). ***JavaScript es sensitivo a mayúsculas y minúsculas***, por tanto, los nombres con diferencias en mayúsculas o minúsculas son nombres de variables distintas.

3.2. DECLARACIÓN DE VARIABLES

Para decir a JavaScript que un identificador se va a utilizar como una variable, ésta se debe declarar primero. Para declarar variables en JavaScript se utiliza la palabra reservada "**var**" seguida del nombre de la nueva variable. Esta acción reserva el nombre como una variable que será utilizada como área de almacenamiento para cualquier tipo de datos que se quieran guardar en ella. Para declarar más de una variable en la misma sentencia se utiliza la coma (,) como separador. La siguiente tabla muestra ejemplos de declaraciones de variables:

Declaraciones
var Dirección;

```
var n;
var i,j, k;
var Prueba, Dato;
```

Una vez que una variable se declara, está preparada para ser rellenada con su primer valor. La inicialización se realiza utilizando el operador de asignación (=). Se puede inicializar una variable al mismo tiempo que se declara o en cualquier otro punto después en el script. Asignar un valor a una variable cuando se declara puede ayudamos a recordar el tipo de valor original para el que se creó la variable.

```
Declaraciones
var Direccion="nombre@compañía";
var n=0.00;
var i=0, j=0, k;
var Prueba="hola", Dato=0;
```

JavaScript lee el código de arriba hacia abajo realizando en cada paso la instrucción leída. Hasta que el programa alcanza un paso en el que se inicializa una variable esta permanece indefinida *(undefined)* y de ella no se puede extraer ninguna información. Intentar obtener un valor de una variable antes de que ésta se inicialice causaría un error en la aplicación cuando se ejecuta.

JavaScript ofrece otra posibilidad de declarar una variable: simplemente inicializándola y sin utilizar la palabra reservada "var". Asignar un valor a una nueva variable sin haberla declarado antes con "var", hace que JavaScript automáticamente la declare. Aunque esto puede acelerar la escritura de programas, es una buena práctica de programación declarar todas las variables específicamente.

3.3. TIPOS DE VARIABLES

Cuando se almacena información, automáticamente se la clasifica en uno de los cinco tipos de datos válidos en JavaScript. La siguiente tabla muestra los diferentes tipos de datos utilizados en JavaScript:

Tipos de datos	**Ejemplo**
Number	-19, 3.141516

Boolean	true, false
String	"Buenos días mundo",""
function	unescape, write
object	window, document, null

Una variable de tipo ***number*** almacena un número entero o un número real. Una variable de tipo ***boolean*** mantiene el valor *"true"* o el valor *"false"*. Las variables de tipo ***string*** pueden almacenar cualquier literal de cadena que se les asigne incluso la cadena vacía. En la tabla anterior se muestra cómo representar la cadena vacía con dos dobles comillas. Las funciones *(tipo **function**)* pueden definirse por el usuario o pertenecer al lenguaje. Por ejemplo, la función *"unescape"* está definida en JavaScript. Más adelante se explica la forma de definir nuestras propias funciones. Las funciones que pertenecen a los objetos, denominadas *métodos* en JavaScript, también se clasifican bajo el tipo de datos *function.* Elementos tales como *window* o *document* tienen un tipo de datos ***object.*** Las variables de tipo *object, o* simplemente "objetos", pueden contener otros objetos en su interior. Una variable que contiene en su interior el valor *null* se dice que tiene un tipo de datos *object.* Esto se debe a que JavaScript clasifica el valor *null* como un objeto.

JavaScript es un lenguaje **débilmente tipado**. No se requiere que se definan los tipos de datos y no se impide que se asignen diferentes tipos de datos a una misma variable. Las variables de JavaScript pueden aceptar un nuevo tipo de datos en cualquier momento, en cuanto cambia el tipo de contenido de la variable.

3.4. ÁMBITO DE LAS VARIABLES

El *ámbito* de una variable hace referencia al área o áreas dentro del programa en las cuales la variable puede ser referenciada. Suponiendo que insertamos un script en la cabecera de un documento HTML y otro distinto dentro del cuerpo del mismo documento HTML, JavaScript considera que las variables declaradas en cualquiera de estas áreas tienen el mismo ámbito. Estas variables se consideran ***globales***, y son accesibles por cualquier script en el documento actual. Las variables declaradas dentro de las funciones se consideran ***locales*** y no son accesibles por cualquier script.

Una variable declarada dentro de una función tiene un ámbito local. Solamente esa función puede acceder al valor que mantiene dicha variable. Cada vez que se llama a la función, la variable se crea. De igual forma, cada vez que la función finaliza, la

variable se elimina. Cuando varias funciones declaran una variable con el mismo nombre, JavaScript las considera variables diferentes. Cada una tendrá su dirección y su bloque de memoria.

Si se desea que varias funciones compartan una variable, hay que declarar dicha variable fuera de cualquier función (pero, por supuesto, dentro de las etiquetas *<SCRIPT>)*. De esta forma, cualquier parte de la aplicación, incluyendo todas las funciones, pueden compartir la variable.

IMPORTANTE: Ámbito de las variables

Scope in JavaScript

A local variable can have the same name as a global variable, but it is entirely separate; changing the value of one variable has no effect on the other. Only the local version has meaning inside the function in which it is declared.

JavaScript

```
// Global definition of aCentaur.
var aCentaur = "a horse with rider,";

// A local aCentaur variable is declared in this function.
function antiquities(){

   var aCentaur = "A centaur is probably a mounted Scythian warrior";
}

antiquities();
aCentaur += " as seen from a distance by a naive innocent.";

document.write(aCentaur);

// Output: "a horse with rider, as seen from a distance by a naive innocent."
```

In JavaScript variables are evaluated as if they were declared at the beginning of whatever scope they exist in. Sometimes this results in unexpected behaviors.

JavaScript

```
var aNumber = 100;
tweak();

function tweak(){

    // This prints "undefined", because aNumber is also defined locally
below.
    document.write(aNumber);

    if (false)
```

```
    {
        var aNumber = 123;
    }
}
```

When JavaScript executes a function, it first looks for all variable declarations, for example `var someVariable;`. It creates the variables with an initial value of **undefined**. If a variable is declared with a value, for example `var someVariable = "something";` then it still initially has the value **undefined**, and takes on the declared value only when the line containing the declaration is executed.

Otro modo de ver esto.(anidamientos)

If you're in the global scope then there's no difference.

If you're in a function then "var" will create a local variable, "no var" will look up the scope chain until it finds the variable or hits the global scope (at which point it will create it):

```
// These are both globals
var foo = 1;
bar = 2;

function()
{
    var foo = 1; // Local
    bar = 2;     // Global

    // Execute an anonymous function
    (function()
    {
        var wibble = 1; // Local
        foo = 2; // Inherits from scope above (creating a closure)
        moo = 3; // Global
    }())
}
```

If you're not doing an assignment then you need to use var:

```
var x; // Declare x
```

3.5. VECTORES

Estos tipos de datos complejos son un conjunto ordenado de elementos, cada uno de los cuales es en sí mismo una variable distinta. En JavaScript, los vectores y las matrices son objetos. Como veremos que hacen todos los objetos, se declaran utilizando el operador new:

miEstupendoVector = new Array(20)

El vector tendrá inicialmente 20 elementos (**desde 0 hasta 19**). Si queremos ampliarlo no tenemos más que asignar un valor a un elemento que esté fuera de los límites del vector:

miEstupendoVector[25] = "Algo"

De hecho, podemos utilizar de índices cualquier expresión que deseemos utilizar. Ni siquiera necesitamos especificar la longitud inicial del vector si no queremos:

vectorRaro = new Array();

vectorRaro["A colocar en los bookmark"] = "HTML en castellano";

4. OPERADORES

Los operadores nos permiten unir identificadores y literales para formar expresiones. Las expresiones son el resultado de operaciones matemáticas o lógicas. Un literal o una variable son expresiones, pero también lo son esos mismos literales y variables unidos entre sí mediante operadores.

JavaScript dispone de muchos más operadores que la mayoría de los lenguajes, si exceptuamos a sus padres C, C++ y Java. Algunos de ellos no los estudiaremos debido a su escasa utilidad y con algunos otros (especialmente el condicional) deberemos andarnos con cuidado, ya que puede lograr que nuestro código no lo entendamos ni nosotros.

4.1. OPERADORES ARITMÉTICOS

JavaScript dispone de los operadores aritméticos clásicos y algún que otro más:

Descripción	**Símbolo**	**Expresión de ejemplo**	**Resultado del ejemplo**
Multiplicación	*	2*4	8
División	/	5/2	2.5
Resto de una división entera	%	5 % 2	1
Suma	+	2+2	4
Resta	-	7-2	5
Incremento	++	++2	3
Decremento	--	--2	1

Menos unario	-	-(2+4)	-6

Los operadores de incremento y decremento merecen una explicación auxiliar. Se pueden colocar tanto antes como después de la expresión que deseemos modificar pero sólo devuelven el valor modificado si están delante. Me explico.

a = 1; b = ++a;

En este primer caso, a valdrá 2 y b 2 también. Sin embargo:

a = 1; b = a++;

Ahora, a sigue valiendo 2, pero b es ahora 1. Es decir, estos operadores modifican siempre a su operando, pero si se colocan detrás del mismo se ejecutan después de todas las demás operaciones.

4.2. OPERADORES DE COMPARACIÓN

Podemos usar los siguientes:

Descripción	**Símbolo**	**Expresión de ejemplo**	**Resultado del ejemplo**
Igualdad	= =	2 = = '2'	Verdadero
Desigualdad	!=	2 != 2	Falso
Menor que	<	2 < 2	Falso
Mayor que	>	3 > 2	Verdadero
Menor o igual que	<=	2 <= 2	Verdadero
Mayor o igual que	>=	1 >= 2	Falso

Hasta la versión 1.1 de JavaScript al comparar un número (3) con una cadena (“3”), JavaScript convertía el operando de tipo cadena en un número y después realizaba la comparación, encontrando en este caso a ambos operandos iguales. A partir de la versión 1.2 de JavaScript, es una tarea del programador convertir los dos operandos al mismo tipo para que puedan ser comparados. En el ejemplo dado, sin realizar conversiones, los dos operandos serán diferentes.

4.3. OPERADORES LÓGICOS

Estos operadores permiten realizar expresiones lógicas complejas:

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Negación	!	!(2 = 2)	Falso
Y	&&	(2 = 2) && (2 >= 0)	Verdadero
Ó	||	(2 = 2) || (2 <> 2)	Verdadero

4.4. OPERADORES DE ASIGNACIÓN

Normalmente los lenguajes tienen un único operador de asignación, que en JavaScript es el símbolo =. Pero en este lenguaje, dicho operador se puede combinar con operadores aritméticos y lógicos para dar los siguientes:

Operador	Significado	Operador	Significado
x += y	x = x + y	x -= y	x = x - y
x /= y	x = x / y	x *= y	x = x * y
x % y	x = x % y		

4.5. OPERADORES ESPECIALES

Vamos a incluir en este apartado operadores que no hayan sido incluidos en los anteriores. La concatenación de cadenas, por ejemplo, se realiza con el símbolo +. El **operador condicional** tiene esta estructura:

condicion ? valor1 : valor2

Si la condición se cumple devuelve el primer valor y, en caso contrario, el segundo. El siguiente ejemplo asignaría a la variable a un 2:

a = 2 > 3 ? 1 : 2

Para tratar con objetos disponemos de dos operadores:

Descripción	Símbolo	Expresión de ejemplo	Resultado del ejemplo
Crear un objeto	new	a = new Array()	a es ahora un vector
Borrar un objeto	delete	delete a	Elimina el vector anteriormente creado

5. ESTRUCTURAS DE CONTROL

Las estructuras de control nos permiten modificar el flujo de ejecución básico del *script*, es decir, gracias a ellas la ejecución del *script* no tiene por qué ser totalmente lineal. También nos permiten que ciertas partes del *script* no tengan que ejecutarse siempre.

5.1. SENTENCIAS CONDICIONALES

A través de las sentencias *if* y *else* somos capaces de tomar una decisión basándonos en una expresión y hacer que el programa elija entre dos caminos de ejecución diferentes.

5.1.1. If

La sintaxis de la sentencia *if es* la siguiente:

if (condición) {
[sentencias]
}

La *condición* debe ser una **expresión lógica**. Si el resultado de evaluar la *condición* es *true,* las sentencias se ejecutan, y continúa la ejecución del programa. Si el resultado de la *condición* es *false,* JavaScript ignora las sentencias y continúa.

En el siguiente ejemplo, podemos observar la utilización de la sentencia *if* para saber cuál es el mayor de un conjunto de tres valores. Para descubrir dicho valor, utilizamos tres sentencias *if,* cada una de las cuales, a través de una expresión lógica, pregunta si un valor determinado es mayor que los otros dos:

Ejem5.html

```
<HTML>
<HEAD><TITLE>If</TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE ="JavaScript">
<!-- se oculta la información de los navegadores antiguos
var datol = 15;
var dato2 = 9;
var dato3 = 17;
document.write("<H3> El mayor valor de ");
document.write(+datol+", "+dato2+" y "+dato3+" es:</H3>");
if ((datol>dato2)&&(datol>dato3)) {
document.write("<Hl>"+datol+"</Hl>"); }
```

```
if ((dato2>datol)&&(dato2>dato3)) {
document.write("<Hl>"+dato2+"</Hl>"); }
if ((dato3>datol)&&(dato3>dato2)) {
document.write("<H1>"+dato3+"</H1>");}
document.write("<H3>...Hecho</H3>");
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY>
</HTML>
```

5.1.2. If...else

A veces, la utilización de la sentencia *if en* solitario no es suficiente, porque se necesita ejecutar ciertas sentencias cuando la expresión condicional no se cumpla. Para poder hacerlo, añadimos un bloque *else* al bloque *if,* con lo que la sintaxis de esta sentencia *if...else* queda de la siguiente forma:

```
if (condición) {
[sentencias]
} else {
[sentencias]
}
```

Si la expresión lógica que reside en la *condición* se evalúa como *true* se ejecutarán las sentencias del bloque *if,* mientras que si se evalúa como *false* se ejecutarán las sentencias del bloque *else.*

El siguiente ejemplo es una versión del código que obtenía el mayor de tres valores, utilizando sentencias *if...else:*

Ejem7.html

```
<HTML>
<HEAD> <TITLE>If-Else </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE ""JavaScript">
<!-- se oculta la información de los navegadores antiguos
```

```
var datol =15;
var dato2 = 9;
var dato3 = 17 ;
document.write("<H3> El mayor valor de ");
document.write(+datol+", "+dato2+" y "+dato3+" es:</H3>");
if (datol>dato2){
        if(datol>dato3) {
        document.write("<H1>"+datol+"</Hl>") ;
        } else {
                document.write("<H1>"+dato3+"</H1>") ;}
} else  {
        if (dato2>dato3){
        document.write("<Hl>"+dato2+"</Hl>") ;
        } else {
                document.write("<Hl>"+dato3+"</Hl>") ;}
        }
document.write("<H3>...Hecho</H3>") ;
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY></HTML>
```

5.2. BUCLES

La utilización de bucles dentro de un script sirve para muchos propósitos. Un uso sencillo, pero habitual, es utilizar los bucles para contar. También se utilizan para recorrer objetos o estructuras compuestas por más de un elemento, como ocurre con las estructuras de tipo *array*.

5.2.1 for

El bucle *for* tiene la siguiente sintaxis:

```
for ([exp_inicialización]; [exp_condición];[exp_bucle]){
[sentencias]
}
```

Las tres expresiones cerradas entre paréntesis son opcionales, pero es necesario escribir los caracteres punto y coma (;) aunque se omitan, para que cada expresión permanezca en el lugar apropiado. Normalmente se utiliza la *exp_inicialización* para inicializar y declarar la variable que se utiliza como contador del bucle. La expresión *exp_condición* define la condición que ha de cumplirse y ha de ser evaluada a *true*, para poder ejecutar las sentencias encerradas entre las llaves.

Finalmente la expresión *exp_bucle*, incrementa o decrementa la variable utilizada como contador del bucle.

El siguiente ejemplo muestra la utilización del bucle *for* para imprimir los números pares comprendidos entre el 1 y el 100.

Ejem8.html

```
<HTML>
<HEAD> <TITLE>For  </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE="JavaScript">
<!--se  oculta la Información de los navegadores antiguos
document.write("<H3>Números pares entre 1 y100 ...</H3>");
        for (i=1; i<=100; i++){
                if (i%2==0){
                        document.write (i+", ");
                }
                if (i%10==0){ document.write ("<BR>"); }
        }
        document.write ("<h3> ... hecho</h3>");
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY></HTML>
```

El ejemplo utiliza una sentencia *if* que produce una nueva línea cuando el contador del bucle es un número múltiplo de 10.

Al igual que las sentencias *if*, las sentencias *for* se pueden anidar.

Ejem9.html

```
<HTML>
<HEAD><TITLE>For anidado</TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE =JavaScript>
<!-- se oculta la información de los navegadores antiguos
var sumatorio=0;
document.write ("<H3>Sumatorio de cada valor entre 1 y 100...</H3>");
for (i=1 ; i<=100; i++){
        for (j=i ; j>0 ; j--){
                sumatorio+=j;
        }
        document.write(sumatorio +", ");
        if (i%10==0) {document.write("<BR>");  }
        sumatorio=0;
}
document.write("<H3>...Hecho</H3>");
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY>   </HTML>
```

5.2.2 for...in

El bucle *for...in* tiene la siguiente sintaxis:

for (var propiedad in objeto) {
[sentencias]
}

Este bucle se utiliza para ejecutar un conjunto de sentencias sobre cada una de las propiedades de un objeto. Produce una iteración a través de todas las propiedades de un objeto. Para utilizarlo, por tanto, hace falta conocer el modelo

de objetos de JavaScript, que se explica en el capítulo siguiente. Aunque lo explicamos en esta sección por ser una estructura de control.

En la sintaxis, *propiedad* es una variable que en cada iteración del bucle es asignada a una propiedad del *objeto,* hasta que se llega a la última propiedad del *objeto.* Si éste no posee propiedades no se produce ninguna iteración.

El siguiente ejemplo muestra la utilización de este tipo de bucle:

Ejem10.html

```
<HTML>
<HEAD> <TITLE>For in </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE ="JavaScript">
<!-- se oculta la información de los navegadores antiguos
document.write("<H3>Propiedades  objeto DOCUMENT ...</H3> ");
for (var propiedad in document){
document.write("document." + propiedad + " = " + document
[propiedad]);
document.write("<br>") ;
}
document.write("<H3>...Hecho</H3>");
// final del comentario -->
</SCRIPT>>
</CENTER>
</BODY></HTML>
```

5.2.3 while

La sentencia ***while*** actúa de forma muy parecida a la sentencia for, pero se diferencia de ésta en que no incluye en su declaración la inicialización de la variable de control del bucle ni su incremento o decremento. Por tanto, dicha variable se deberá declarar antes del bucle *while* y su incremento o decremento se deberá realizar dentro del cuerpo de dicho bucle. Su sintaxis es la siguiente:

```
while (condición) {
[sentencias]
```

```
}
```

Si la *condición* se evalúa a *true* se ejecutan las sentencias del cuerpo del bucle; después de ejecutarlas se volverá a evaluar la *condición*, de forma que si ésta sigue cumpliéndose se volverán a ejecutar las sentencias. Si la *condición* se evalúa a *false* no se ejecutarán las sentencias del cuerpo del bucle. El siguiente ejemplo muestra la utilización del bucle *while* para obtener los números pares entre *1 y 100* de igual forma que en el ejemplo del bucle *for:*

Ejem11.html

```
<HTML>
<HEAD>  <TITLE>While </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE ="JavaScript">
<!--se  oculta la Información de los navegadores antiguos
var i=1;
document.write("<H3>Números pares entre 1 y100 ...</H3>");
while ( i<=100){
if (i%2==0){  document.write(i+", "); }
        if (i%10==0){ document.write("<BR>"); }
        i++;
}
document.write("<h3> ... hecho</h3>");
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY></HTML>
```

De igual forma que las sentencias anteriores, los bucles *while* se pueden anidar.

5.2.4 do...while

A partir de la versión 1.2, JavaScript ofrece una nueva estructura de control: el *do...while.* Funciona exactamente igual que el *while,* excepto que la

condición no se comprueba hasta que se ha realizado una iteración. Esto garantiza que al menos el cuerpo del bucle se realiza una vez. Su sintaxis es la siguiente:

do{
[sentencias]
} while (condición)

El siguiente código es un ejemplo de la utilización del bucle *do...while*. El programa mostrará al menos los 25 primeros números pares y después, en función de la respuesta del usuario a la pregunta: *¿Deseas ver los 25 próximos pares?*, que se realiza a través de la función ***confirm()*** de JavaScript, se finalizará el programa (*Cancelar)* o se mostrarán los siguientes 25 números pares (*Aceptar).* Después de mostrar cada grupo de 25 pares, el programa vuelve a realizar la pregunta anterior, que es la condición de salida del bucle *do...while:*

Ejem13.html

```
<HTML>
<HEAD><TITLE>Do…While </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE="JavaScript">
<!--se  oculta la Información de los navegadores antiguos
var k=0;
document.write("<H3>Números pares entre 1 y ...</H3>");
do {
        k=k+50;
        for (i=k+1-50; i<=k; i++){
                if (i%2==0){
                        document.write(i+", ");
                }
                if (i%10==0){ document.write("<BR>"); }

        }
} while (confirm("¿Deseas ver los 25 próximos pares?"));
document.write("<H3> ... hecho</H3>");
```

```
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY></HTML>
```

5.2.5 break y continue

A veces es necesario salir de un bucle antes de que la condición del mismo se evalúe a *falso*, o incluso antes de alcanzar cierta posición dentro del cuerpo del bucle. Para ello utilizamos las sentencias *break*, que finaliza la ejecución del bucle y *continue*, que salta las sentencias posteriores a ella y evalúa de nuevo la expresión del bucle (si existe) continuando con la siguiente iteración.

El siguiente ejemplo nos muestra la utilización de las sentencias *break* y *continue*. El programa obtiene los sumatorios (suma de todos los números comprendidos entre un número dado y el *0)* con valor par e inferior a 1000 de los números comprendidos entre 1 y 100. La sentencia *break* se utiliza para finalizar el bucle principal una vez que se encuentra el primer sumatorio con valor superior a 1000 y la sentencia *continue* para mostrar sólo los sumatorios con resultados impares:

Ejem14.html

```
<HTML>
<HEAD>  <TITLE>Break y continue </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE=JavaScript>
<!-- se oculta la información de los navegadores antiguos
var sumatorio=0;
document.write ("<H3>Sumatorios impares inferiores a 1000.. .</H3>");
for (i=1 ; i<=100; i++){
        sumatorio=0;
        for (j=i ; j>0 ; j--){
                sumatorio+=j;
        }
```

```
        if (i%8==0) {document.write ("<BR>");}
        if (sumatorio>1000){break;}
        if (sumatorio%2==0){continue;}
document.write(sumatorio +", ");
} document.write("<H3>...Hecho</H3>");
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY>    </HTML>
```

5.3 *SENTENCIA WITH*

La sentencia *with* se utiliza para evitar la escritura repetida de la especificación de la referencia a un objeto cuando se está accediendo a las propiedades o métodos de dicho objeto. Su sintaxis es la siguiente:

```
with (objeto) {
[sentencias]
}
```

El *objeto* especifica la referencia al objeto que se está utilizando. Y que debería de aparecer en algunas de las sentencias del cuerpo. El siguiente ejemplo nos muestra la utilización de la sentencia *with* para obtener información del documento de trabajo:

Ejem15.html

```
<HTML>
<HEAD>  <TITLE> With </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE="JavaScript">
<!-- se oculta la información de los navegadores antiguos
   with (document) {
        write("<H3>Utilizando la sentencia \"WITH\" ...</H3>");
        write("título: \""+ title +  "\"<BR>");
        write("<BR>");
        write("color de fondo:" + bgColor + "<BR>");
```

```
        write("color de primer plano:" + fgColor + "<BR>");
        write("color de los enlaces visitados:" + vlinkColor + "<BR>");
        write("color de los enlaces no visitados:" + alinkColor + "<BR>");
        write("<H3>...Hecho</H3>");
}
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY>  </HTML>
```

5.4 SENTENCIA SWITCH

La sentencia ***switch*** se utiliza para comparar un dato entre un conjunto de posibles valores. Esta tarea se puede realizar utilizando múltiples sentencias *if*, pero la sentencia *switch* es mucho más legible y nos permite especificar un conjunto de sentencias por defecto, en el caso que el dato no tenga un valor con que compararlo. La sintaxis de la sentencia *switch* es la siguiente:

```
switch (datoAcomparar){
case valor 1: [sentencias]
case valor2: [sentencias]
case valor3: [sentencias]
case valorn: [sentencias]
default: [sentencias]
}
```

El siguiente ejemplo muestra la utilización de la sentencia *switch* para discriminar entre un conjunto de usuarios:

Ejem16.html

```
<HTML>
<HEAD> <TITLE>Switch </TITLE></HEAD>
<BODY>
<CENTER>
<SCRIPT LANGUAGE=JavaScript>
<!-- se oculta la información de los navegadores antiguos
var usuario="";
```

```
document.write ("<h3>Uso de la sentencia \"SWITCH\" ...</H3>");
usuario=prompt ("Introduce el usuario de acceso:","");
switch (usuario.toUpperCase()){
case "CARLOS": document.write("Bienvenido Carlos ...<BR>"); break;
case "JUAN": document.write("Bienvenido Juan ...<BR>");   break;
case "PEPE": document.write("Bienvenido Pepe ...<BR>");    break;
default:  document.write("No eres un usuario autorizado ...<BR>");  break;
}
document.write ("<H3>...Hecho</H3>");
// final del comentario -->
</SCRIPT>
</CENTER>
</BODY></HTML>
```

Inicialmente y a través de la función *propmt()* de JavaScript se le solicita al usuario que introduzca su nombre. Esta función muestra el siguiente cuadro de diálogo que permite al usuario introducir información, que se devuelve para ser almacenada en la variable *usuario.*

Después la información introducida se compara secuencialmente con los tres usuarios predefinidos *(CARLOS, JUAN* y *PEPE)* hasta encontrar la igualdad con un de ellos y realizar un saludo personalizado. En caso de que el usuario introducido no sea ninguno de los tres preestablecidos se ejecutan las sentencias de la cláusula *default.* Si no se utiliza la sentencia *break* dentro de las cláusulas *case,* la ejecución de una cláusula continúa por la siguiente.

En el código del ejemplo se utiliza el método *toUpperCase()* asociado con las cadenas de caracteres, cuya función es pasar todas las letras de una cadena a mayúsculas, para comparar la información introducida con los valores de los nombres en mayúsculas, evitando de este modo que "*Pepe" y "PEPE"* se consideren usuarios diferentes.