MANUAL DE ESTÁNDARES DE DESARROLLO WEB

MANUAL DE DESARROLLO EN JAVASCRIPT

Junta de Castilla y León

Coordinador de la Información: Área de Desarrollo y Mantenimiento
Dirección General de Telecomunicaciones y Transportes.

ÍNDICE DE CONTENIDOS

1 Normativa JavaScript 3

Introducción al JavaScript 4
- Versiones de JavaScript 5
- JavaScript y la especificación ECMA 5
- JavaScript y Java 6

Normas Generales de Codificación en JavaScript 8
- Sangrado 8
- Comentarios 9
- Declaraciones 12
- Sentencias 13
- Uso de espacios en blanco 15
- Nombrado 16
- Posicionamiento del código JavaScript 17

Recomendaciones Generales de Codificación en JavaScript 19
- Versión del lenguaje 19
- Ocultación del código JavaScript 20
- Uso de caracteres especiales 20
- Uso del tag <NOSCRIPT> 21
- Uso de links JavaScript 21
- Entidades JavaScript 21
- Apertura de ventanas popup 22

Tratamiento de errores 25
- Tipos de errores 25
- Consolas de error 26
- Errores frecuentes 27

Seguridad en JavaScript 31
- Marcado de datos (data tainting) 31
- Firmado de scripts 31

Referencias 33

CAPITULO

1 Normativa JavaScript

Este documento pretende describir una serie de estándares para escribir código JavaScript robusto y fácil de mantener.

Este documento no pretende servir como referencia del lenguaje como tal, ni ser un manual de aprendizaje. Se presupone por parte del lector un conocimiento básico de la sintaxis y codificación en JavaScript.

Se establecerán unas normas para codificar en JavaScript y se recogerán una serie de convenciones y sugerencias de codificación a seguir por la comunidad de programadores. Cubre muchos aspectos, desde los nombres de los ficheros, a la organización de éstos, comentarios, declaraciones, sentencias, espacios en blanco, convenciones de nomenclatura, prácticas de programación. Se incluye código de ejemplo que ilustra cada uno de los puntos descritos.

Introducción al JavaScript

El lenguaje JavaScript es un lenguaje multiplataforma de scripting creado por Netscape con el objetivo de extender las funcionalidades del HTML (**H**yper**T**ext **M**arkup **L**anguage) y dotar de mayor dinamismo a las páginas web.

JavaScript está compuesto por un núcleo del lenguaje que contiene:

- Palabras clave del lenguaje.
- Reglas de sintaxis y gramática de las sentencias.
- Reglas referentes a expresiones, variables y literales.
- Un modelo de objetos base.
- Funciones y objetos predefinidos (Array, Date, Math).

Este núcleo del lenguaje se puede extender con objetos adicionales para obtener más funcionalidades. Existen dos extensiones del núcleo de JavaScript.

JavaScript cliente.

Extiende el núcleo del lenguaje proporcionando objetos para controlar el navegador y su Modelo de Objetos del Documento (*DOM – **D**ocument **O**bject **M**odel*).

El JavaScript cliente es interpretado por los navegadores web o browsers (como por ejemplo, Internet Explorer o Netscape Navigator). Las sentencias de JavaScript cliente están contenidas en la propia página HTML. Cuando un browser hace una petición de una página, el servidor envía el contenido completo del documento (incluidas las sentencias JavaScript), a través de la red, hacia el cliente. Ya es el propio cliente el que se encarga de interpretar tanto el HTML como el JavaScript.

JavaScript servidor.

Extiende el núcleo del lenguaje proporcionando objetos que permiten la ejecución de JavaScript en un servidor. De esta manera se permite funcionalidades como conexión con una base de datos relacional o la manipulación de ficheros en el servidor.

Las páginas HTML que contienen JavaScript servidor son compiladas en ficheros de bytecodes ejecutables. Estos ficheros son ejecutados en servidores web que contengan una motor JavaScript, como el Netscape Enterprise Server.

En adelante sólo se hará referencia al uso de JavaScript modo cliente. El JavaScript en modo servidor se encuentra en desuso, siendo aconsejable utilizar tecnologías de servidor mucho más evolucionadas y robustas.

Versiones de JavaScript

El lenguaje JavaScript ha evolucionado desde su creación a través de diferentes versiones. La primera versión JavaScript 1.0 nación con Netscape 2.0. A partir de aquí han surgido diferentes versiones soportadas por diferentes versiones de los navegadores

A continuación se muestra una tabla con las diferentes versiones de JavaScript y los navegadores que la soportan:

Navegador	Versión JavaScript
NAV 2.0	1.0
NAV 3.0	1.1
NAV 4.0 – 4.05	1.2
NAV 4.06	1.3
NAV 6	1.5
MIE 3.0	Jscript
MIE 4.0 o superior	Estándar ECMA

JavaScript y la especificación ECMA

ECMA (***E**uropean **C**omputer **M**anufactures **A**ssociation*) es una asociación internacional de industrias con base en Europa, fundada en 1961, dedicada a la normalización de los sistemas de comunicación y de información. (*http://www.ecma.ch*) . Forman parte de la asociación en calidad de miembros ordinarios sociedades como las mismísimas Netscape y Microsoft, Sun, IBM, Compaq, Philips, Hewlett-Packard e Intel.

Netscape Communications presentó JavaScript 1.1 a la ECMA para su normalización en otoño de 1996. La Asamblea General de la ECMA presentó en junio de 1997 el estándar definido como ECMAScript o ECMA-262.

Microsoft adoptó este estándar y lo incorporó a partir de su navegador Internet Explorer 4.0.

Sin embargo Netscape avanzó en las versiones de JavaScript de forma independiente. La siguiente tabla describe la relación entre JavaScript y las versiones ECMA.

Versión JavaScript	Relación con ECMA

JavaScript 1.1.	ECMA-262 está basado en JavaScript 1.1
JavaScript 1.2.	ECMA-262 no estaba completado cuando apareció JavaScript 1.2. Por lo tanto no son totalmente compatibles por las siguientes razones: • JavaScript 1.2 incorpora características adicionales que no están consideradas en ECMA-262 • ECMA-262 añade dos características: Internacionalización usando UNICODE, y comportamiento uniforme entre plataformas. JavaScript 1.2 tiene varias características dependientes de plataforma como el objeto Date. .
JavaScript 1.3	JavaScript 1.3 es totalmente compatible con ECMA-262., pero incorpora características adicionales no contempladas en ECMA-262

En Junio de 1999 ECMA presentó una nueva versión de ECMAScript, ECMA-290.

JavaScript y Java

JavaScript y Java son similares en algunos aspectos, pero fundamentalmente difieren en otros. El lenguaje JavaScript se asemeja al lenguaje Java, sin embargo no tiene ni el tipado estático ni el control fuertemente tipado.

JavaScript soporta la mayoría de las expresiones sintácticas y estructuras básicas de control de flujo de Java.

Java es un sistema de precompilación de clases construidas por declaración y, JavaScript soporta un sistema de compilación en tiempo de ejecución basado en un número pequeño de tipos de datos representados en numérico, boolean y cadenas. JavaScript tiene un prototipo basado en el modelo de objetos. El prototipo le provee de herencia dinámica. JavaScript también soporta funciones sin unos requisitos declarativos concretos.

JavaScript es un lenguaje de sintaxis mucho más libre que Java, ya que no es necesario declarar todas las variables, clases y métodos. No es necesario especificar la accesibilidad de los métodos, ni el tipo de los valores de retorno.

A continuación aparece un cuadro comparativo entre Java y JavaScript:

JavaScript	Java
Interpretado por el cliente, no compilado.	Compilado en bytecodes descargados desde el servidor y ejecutados en el

	cliente.
Orientado a objetos. No hay distinción en los tipos de los objetos. La herencia es a través del mecanismo de prototipos y los atributos y métodos pueden ser añadidos a cualquier objeto dinámicamente.	Orientado a objetos. Los objetos son divididos en clases e instancias con toda la herencia dada por la jerarquía de clases. A las clases e instancias no se les pueden añadir atributos y métodos dinámicamente.
Código integrado y embebido en HTML	Applets distinto desde HTML.
El tipo de dato de la variable no se declara.	El tipo de dato de la variable se declara.
La referencias de los objetos se comprueban en tiempo de ejecución	Las referencias de los objetos deben existir en tiempo de compilación.
No puede escribir automáticamente en el disco duro.	No puede escribir automáticamente en el disco duro.

Normas Generales de Codificación en JavaScript

Sangrado

Unidad de sangrado

La unidad de sangrado está formada por 4 espacios, no permitiéndose el uso de tabuladores[1] (su representación puede variar dependiendo de editores).

Longitud de líneas

Las líneas no deben ocupar más de 80 caracteres. En ciertos terminales o editores no se ven correctamente si su longitud es mayor.

Expresiones en varias líneas

Cuando una expresión es demasiado larga para que ocupe una única línea, su separación en varias líneas debe ajustarse a las siguientes reglas:

- Cortar tras una coma

Ejemplo:

```
funcion1(parametro1, parametro2,
     parametro3);
```

- Cortar antes de un operador

Ejemplo:

```
var resultado = expresion1 * (expresion2 + expresion3)
                  - expresion4
```

- El corte debe situarse en el menor nivel de profundidad posible

Ejemplo:

```
var resultado = expresion1 * (expresion2
          + expresion3) - expresion4 //EVITAR

var resultado = expresion1 * (expresion2 + expresion3)
          - expresion4 //CORRECTO
```

[1] La tabulación hace referencia al carácter de tabulación, no al uso de la tecla de tabulación, siempre que ésta escriba espacios y no tabuladores.

- Alinear el principio de la nueva línea con el principio de la subexpresión cortada en la línea anterior

Ejemplo:

```
var resultado = expresion1 * (expresion2
                            + expresion3)
```

- Si la regla anterior produce un código confuso debido a que la sangría es demasiado grande, se puede cambiar por sangrado de 8 espacios.

- Para las sentencias condicionales la sangría debería ser de 8 caracteres en lugar de alinearla con el principio de la subexpresión

Ejemplo:

```
  /* En este bloque no se distingue
  * la sentencia de las
  * condiciones
  */
 if (condicion1
     || condicion2
     || condicion3) {
     ejecutar();
  }


 /* En este sí */
 if (condicion1
       || condicion2
       || condicion3) {
     ejecutar();
 }
```

Comentarios

Normas básicas

Antes de mostrar el formato para los comentarios en JavaScript, es conveniente señalar varias reglas básicas que hay que seguir para obtener una buena documentación:

- Los comentarios sólo deben contener información relevante para entender el código, evitando incluir información sobre características externas al mismo (ubicación en un directorio, instalación,...).
- Evitar la inclusión de comentarios que queden obsoletos con facilidad.
- Evitar la inclusión de comentarios para duplicar información ya presente en el código de forma explícita.

- La frecuencia de comentarios implica a veces código de baja calidad. Si un fragmento de código necesita ser comentado exhaustivamente quizás sea bueno rescribirlo de forma más clara
- Evitar decorar el código. Hay que evitar en todo momento el uso de cajas de asteriscos o florituras de ese tipo, porque el desarrollador debe escribir un código limpio no bonito, además es una pérdida de tiempo que no aporta nada.
- Escribir el comentario antes de escribir el código. Esto ofrece al desarrollador la oportunidad de recapacitar sobré la funcionalidad del código.
- Comentar por qué se hacen las cosas y no qué se está haciendo. Muchas veces es muy sencillo de saber qué está haciendo el código, sin embargo no es sencillo averiguar el porqué, es decir, qué regla de negocio impulsa a hacer, por ejemplo, una comprobación, y eso es precisamente lo que interesa al desarrollador.

Formato de comentarios

Los comentarios de implementación pueden ser de 4 tipos: bloque, línea, rastreo y final de línea.

- Comentarios de bloque

  Los comentarios de bloque se usan para describir ficheros, funciones, estructuras de datos y algoritmos. Pueden ser usados al principio de cada fichero o antes de cada función, aunque se pueden incluir en otros lugares, dentro de funciones.

  Deben cumplir que:

  - Dentro de una función deben estar sangrados al mismo nivel que el código que describen.
  - Deben estar precedidos por una línea en blanco para separarlos del resto del código.

  Su aspecto debe ser el siguiente:

```
/*
 * Esto es un comentario de bloque.
 */
```

- Comentarios de línea

  Se distinguen varios tipos, aunque para todos ellos se propone la misma notación (//)

  Los comentarios de línea deben:

  - Estar sangrados al mismo nivel que el código que les sigue

- Ser precedidos por una línea en blanco para separarlos del resto del código.
- Si el comentario no puede incluirse en una única línea, se deben usar bloques (según se describe en el apartado anterior de comentarios de bloque).

Un ejemplo puede ser:

```
if (condicion) {

    // Hacemos lo que sea
    ....
}
```

- Comentarios de rastreo o fin de línea

Los comentarios de rastreo son comentarios que aparecen en la misma línea de código que describen. Deben cumplir que:

- Si en una porción de código existe más de un comentario de este tipo, deben estar sangrados al mismo nivel
- Deben estar sangrados lo suficientemente lejos de las sentencias para distinguirse claramente.
- Pueden usarse para comentar varias líneas de texto o fragmentos de código.

Ejemplo1:

```
if (a % 2 == 0) {
    alert('Par');        // Ningún par es primo
} else {
    esPrimo(a);          // Sólo ejecutamos si a es impar
}
```

Ejemplo2:

```
if (condicion) {

    // Ejecutamos la sentencia
    ....
} else {
    funcion2();  // Explicamos por qué
}



//if (condicion) {
//
//    //Esto es una prueba
//    ...
//} else {
//    return false;
//}
```

Declaraciones

Una declaración por línea

Se debe declarar un único componente por línea. El lenguaje JavaScript no obliga a declarar las variables antes de ser utilizadas. Sin embargo no declarar una variable explícitamente puede traer problemas de ámbito de utilización. Es por ello que se hace obligatorio la declaración de las variables.

Ejemplo:

```
var operando1 = 0;

function abrir(){
        var operando 1 = 5; //Esta variable no
                            //sobrescribe a la anterior
                            //ya que es local a la
                            //función
}
```

Inicialización

- Las variables deben ser inicializadas en el mismo lugar en el que son declaradas.
- Excepción: Si el valor inicial de una variable depende de una ejecución posterior a la declaración, la inicialización se realizará tras dicha ejecución (no se deben inicializar variables de forma indiscriminada con valores que no tienen sentido).

Posicionamiento

- Las declaraciones deben estar agrupadas al principio de cada bloque (se entiende bloque como el fragmento de código delimitado por {...}).
- Evitar declaraciones locales con el mismo nombre que otras de nivel superior.

Ejemplo:

```
funcion() {
    var entero1 = 0;

    if (condicion) {
         var entero2 = 0;
         ...
    }
}
```

Declaraciones de funciones

Se deben cumplir las siguientes reglas:

- No debe existir espacio entre el nombre de la función y el paréntesis de apertura de la lista de argumentos.
- La llave abierta debe aparecer al final de la misma línea que la sentencia de la declaración, excepto si la declaración ocupa varias líneas en cuyo caso la llave aparecerá al final de la última línea de la declaración.
- La llave cerrada debe aparecer sola en una línea y sangrada a la altura del inicio de la declaración, excepto si la declaración está vacía, en cuyo caso las llaves abierta y cerrada deben aparecer en la misma línea ({}).
- Las declaraciones de funciones están separadas por una línea en blanco.

Sentencias

Sentencias sencillas

Una línea debe contener una única sentencia sencilla

Ejemplo:

```
i++;     //BIEN
j++;     //BIEN
i++; j++;     //MAL
```

Sentencias compuestas

Una sentencia compuesta es una sentencia que contiene un bloque de sentencias sencillas delimitado por {}. (if ... { sentencias }, for .... { sentencias })

- Las sentencias agrupadas deben estar sangradas un nivel más a la derecha que las sentencias compuestas que las contienen.
- La llave abierta debe estar al final de la línea de inicio de la sentencia compuesta.
- Las llaves deben delimitar cualquier sentencia (aunque sea sencilla), cuando formen parte de una estructura de control (tipo if-else o for). De este modo se evita que, al añadir más sentencias, accidentalmente se introduzcan bugs por olvidar los paréntesis.

if, if-else

Deben escribirse de la siguiente manera:

```
if (condicion) {
    sentencias;
}

if (condicion) {
    sentencias;
} else {
    sentencias;
}
```

Sentencias for

- Una sentencia for debe escribirse de la siguiente manera:

```
for (inicializacion; condicion; actualizacion) {
    sentencias;
}
```

- Una sentencia for vacía (aquella que no necesita un cuerpo ya que su funcionalidad completa se ejecuta entre la inicialización, la condición y la actualización) debería escribirse:

```
for (inicializacion; condicion; actualizacion) ;
```

Sentencias while

Una sentencia while debería escribirse:

```
while (condicion) {
    sentencias;
}
```

Una sentencia while vacía debería escribirse:

```
while (condicion) ;
```

Sentencias do-while

Una sentencia do-while debe escribirse:

```
do {
    sentencias;
} while (condicion);
```

Sentencias switch

Una sentencia switch debe escribirse:

```
switch (condicion) {
case ABC:
    sentencias;
    /* no rompe */
```

```
case DEF:
    sentencias;
    break;

case XYZ:
    sentencias;
    break;

default:
    sentencias;
    break;
}
```

En cada bloque case que no rompa (no tiene break al final) es necesario indicarlo explícitamente para evitar confundirlo con un despiste. Esto se puede hacer escribiendo un comentario del tipo /* no rompe */ en el lugar en el que normalmente debería estar el break.

Cada sentencia switch debe tener una etiqueta default.

Uso de espacios en blanco

Líneas en blanco

Las líneas en blanco mejoran la lectura del código fuente por un humano. Las reglas de su uso son las siguientes:

- Dos líneas de código en blanco se deben usar en los siguientes casos:
  - Entre secciones del código fuente (las secciones son: Comentarios iniciales, declaración de variables globales, declaración de funciones).

- Una línea en blanco debería usarse en los siguientes casos:
  - Entre funciones
  - Entre la declaración de las variables locales de una función y la primera sentencia.
  - Antes de un comentario de bloque o de un comentario de línea.
  - Entre secciones lógicas de una función.

Espacios en blanco

El uso de espacios en blanco cumplirá las siguientes reglas:

- Si a una palabra clave le sigue un paréntesis, estos deben estar separados por un espacio en blanco.
- No se debe usar un espacio en blanco entre el nombre de una función y el paréntesis que abre la lista de argumentos. Esto diferencia las palabras clave de las llamadas.

- Tras las comas que separan argumentos en las listas debe aparecer un espacio en blanco.
- Se deben usar para separar operadores binarios de sus operandos.

Ejemplo:

```
a = c + d;
a = (a + b) / (c + d);
```

- Nunca se deben usar para separar operadores unitarios de sus operandos.

Ejemplo:

```
a++;
b++;
```

- Las expresiones en una sentencia for deben estar separadas por espacios.

Ejemplo:

```
for (expr1; expr2; expr3)
```

Nombrado

A continuación se muestra el estándar para el nombrado de objetos que propuesto.

Tipo de identificador	Reglas de nombrado	Ejemplos
Ficheros de código	Los nombres de los ficheros de código JavaScript deben estar escritos en minúscula.	`libreria1.js`

Funciones	Las funciones deben expresar acciones, por lo que deben ser verbos en infinitivo acompañados normalmente del objeto/objetos de la acción o de un adjetivo que los cualifique. La primera palabra estará escrita en minúsculas y el resto con la primera letra en mayúsculas.	`mostrarDetalleCliente()`
		`editarFormulario()`
		`abrirMinimizado()`

Variables	Los nombres de las variables pueden componerse de una o varias palabras con la primera de ellas en minúsculas y el resto con la primera letra en mayúscula. El nombre de una variable debe tener un uso mnemónico, de forma que una lectura ocasional permita intuir su función. El nombre debe ser además conciso, aunque deben evitarse los nombres de un solo carácter, excepto en las variables de uso temporal y función muy definida (índices, auxiliares,...)	`var i;` `var nombreCliente;`

Posicionamiento del código JavaScript

Todo el código JavaScript que no escriba directamente en el documento debe ser posicionado dentro del campo `<HEAD>`.

Ejemplo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">

<HTML>
<HEAD>
<SCRIPT LANGUAGE="JavaScript">
<!--
function functionName() {
    alert(text);
}

var text = 'Hello World';
//-->
</SCRIPT>
</HEAD>

<BODY>

<SCRIPT LANGUAGE="JavaScript">
<!--
document.write('<FORM>' +
               '<INPUT TYPE="BUTTON" VALUE="Click Here" ' +
               'onClick="functionName()"></FORM>');
//-->
</SCRIPT>

</BODY>

</HTML>
```

Esto asegura que todas las definiciones de la funciones de JavaScript estén cargadas por el navegador antes de que sean necesarias. También facilita mucho el mantenimiento del código ya que el código puede encontrarse fácilmente.

Otras posibles localizaciones para el código JavaScript son:

Manejadores de eventos

```
<A HREF="page.htm" onClick="alert('Hello World')">text link</A>
```

Links

```
<A HREF="javascript:alert('Hello world')">text link</A>
```

Ficheros fuente externos

Las funciones de carácter general que puedan ser reutilizadas en varias páginas HMTL o aplicaciones deben extraerse a un fichero aparte con extensión .js, y almacenarse en el directorio js común dentro de la estructura del site destinado a a contener todos los ficheros js con funciones JavaScript necesarias para la aplicación.

Es conveniente crear librerías .js con funciones de carácter general que puedan ser reutilizadas.

En el `<HEAD>` de las páginas se han de incluir todas las librerías necesarias.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/REC-html40/loose.dtd">

<HTML>
<HEAD>
<SCRIPT SRC="../js/library1.js"></SCRIPT>
<SCRIPT SRC="../js/library2.js"></SCRIPT>
……
……
</HEAD>

<BODY>
</BODY>
</HTML>
```

Recomendaciones Generales de Codificación en JavaScript

Versión del lenguaje

Cuando se utilice una característica propietaria de una determinada versión de JavaScript es necesario especificarlo.

Ejemplo:

```
<SCRIPT LANGUAGE=”JavaScript”>
<!—
// JavaScript 1.0 y superior
// -->
</SCRIPT>

<SCRIPT LANGUAGE=”JavaScript1.1”>
<!—
// JavaScript 1.1 y superior
// -->
</SCRIPT>

<SCRIPT LANGUAGE=”JavaScript1.2”>
<!—
// JavaScript 1.2 y superior
// -->
</SCRIPT>

<SCRIPT LANGUAGE=”JavaScript1.3”>
<!—
// JavaScript 1.3 y superior
// -->
</SCRIPT>
```

Si es necesario dentro de una función acceder a propiedades de objetos y métodos no disponibles en todas las versiones de JavaScript, deben darse diferentes definiciones de la función para las versiones de JavaScript más importantes.

Ejemplo:

```
<SCRIPT LANGUAGE=”JavaScript”>
<!--
function test() {
  alert('This browser does not support JavaScript 1.3');
}
// -->
</SCRIPT>

<SCRIPT LANGUAGE=”JavaScript1.3”>
<!—
function test() {
  //Esta función solo es invocada por navegadores
  //que soportan JavaScript 1.3 y reemplaza a la
  //función anterior
}
// -->
</SCRIPT>
```

Como mínimo es siempre recomendable especificar el atributo `LANGUAJE = "JavaScript"`, para evitar posibles confusiones con otros lenguajes de scripting.

Ocultación del código JavaScript

Se ha de incluir siempre el código JavaScript entre comentarios para ocultar el código a los navegadores antiguos que no soporten JavaScript.

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!--

// -->
</SCRIPT>
```

Con esto no se consigue ocultar el código al usuario si este visualiza el código fuente de la página. Únicamente se consigue que el navegador no muestre el código como un contenido más de la página.

La única manera de ocultar por completo el código es colocarlo en frames no accesibles por el usuario o incluir todo el código en ficheros independientes js.

Uso de caracteres especiales

Se debe utilizar las comillas dobles (") para los atributos HTML y la comilla simple (') para las cadenas de caracteres JavaScript.

Ejemplo:

```
<IMG SRC="picture.gif" WIDTH="890" HEIGHT="25">

<SCRIPT LANGUAGE="JavaScript">
<!--
  document.write('<P>');
// -->
</SCRIPT>
```

Este uso de las comillas se complementa de manera correcta cuando se genera código HTML desde JavaScript.

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
  document.write('<IMG SRC="picture.gif" WIDTH="890">');
// -->
</SCRIPT>
```

Cuando se aniden comillas simples dentro de una sentencia debe utilizarse el carácter de escape (" \ ")

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
  document.write('Don\'t forget to escape apostrophes');
// -->
</SCRIPT>
```

Uso del tag <NOSCRIPT>

Se debe utilizar el tag <NOSCRIPT> para proporcionar una alternativa de texto a aquellos navegadores que tengan deshabilitado el JavaScript.

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
  document.write('<P>&Uacute\;ltima modificaci&oacute;n: ' +
document.lastModified + '<\/P>'); // -->
</SCRIPT>

<NOSCRIPT>
<P>&Uacute;ltima modificaci&oacute;n: Febrero 2001
</NOSCRIPT>
```

Uso de links JavaScript

Intente evitar el uso del protocolo `JavaScript:` como una URL dentro de un link. Si el JavaScript está deshabilitado el link no funcionará. En su lugar utilice el propio JavaScript para sobrescribir la propiedad `href` del link.

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!—
  funtion functionName(){
    alert('Hello world');
  }
// -->
</SCRIPT>

<A REF="default.htm"
onClick="this.href='javascript:functionName()'"> texto del link
</A>
```

Entidades JavaScript

El uso de entidades JavaScript está prohibido debido a que solo están soportadas por Netscape Navigator. El siguiente código dará error en otros navegadores:

```
<HR WIDTH="&{barWidth};%">
```

En su lugar deberá usarse:

```
<SCRIPT LANGUAGE=”JavaScript”>
<!—
  document.write(‘<HR WIDTH = ’ + barWidth + ‘%”>’);
// -->
</SCRIPT>
```

Apertura de ventanas popup

La apertura de ventanas flotantes o popups se realiza a través del método `open()` del objeto `window`.

```
window.open(URL,windowName[,windowFeatures])
```

El parámetro `windowFeatures` es una cadena que contiene un conjunto de características separadas por comas que permiten personalizar el aspecto y funcionalidad de la ventana abierta. A continuación mostramos una tabla con las características más importantes.

Característica	Descripción	Tipo yes/no
alwaysLowered	(JavaScript 1.2) Si está puesta a ‘yes’ se crea una nueva ventana que flota por debajo de otras ventanas, ya esté activa o no.	X
alwaysRaised	(JavaScript 1.2) Si está puesta a ‘yes’ se crea una nueva ventana que flota por encima de otras ventanas, ya esté activa o no	X
dependent	(JavaScritpt 1.2) Si está puesta a ‘yes’ se crea una ventana hija de la ventana actual. La ventana hija se cierra si se cierra la ventana padre.	X
height	(JavaScript 1.0 y 1,1) Especifica la altura de la ventana en píxeles	
innerHeight	(JavaScript 1.2) Especifica la altura en píxeles del área de contenido. Esta característica reemplaza a height, la cual permanece por compatibilidad.	
innerWidth	(JavaScript 1.2) Especifica la anchura en píxeles del área de contenido. Esta característica reemplaza a width, la cual permanece por compatibilidad	

location	Si está puesta a 'yes' se muestra la barra de direcciones.	X
menubar	Si está puesta a 'yes' se muestra el menú del navegador	X
resizable	Si está puesta a 'yes' permite al usuario redimensionar la ventana	X
screenX	(JavaScript 1.2) Especifica la distancia de la nueva ventana desde la parte izquierda de la pantalla	
screenY	(JavaScript 1.2) Especifica la distancia de la nueva ventana desde la parte superior de la pantalla	
scrollbars	Si está puesta a 'yes' se crean barras de scroll verticales y horizontales cuando el documento se hace mayor que la ventana	X
status	Si está puesta a 'yes' se muestra la barra de estado del navegador en la parte inferior	X
toolbar	Si está puesta a 'yes' se muestra la barra de herramientas estándar del navegador	X
width	Especifica la anchura de la ventana en píxeles	

Todas las características de tipo 'yes/no' toman el valor 'yes' por defecto en el caso de no utilizar el parámetro `windowFeatures`. Sin embargo si se proporciona el parámetro `windowFeatures` el valor por defecto de las no especificadas es 'no'.

A la hora de realizar aperturas de ventanas flotantes hay que tener en cuenta algunos aspectos.

Para poder tener control de la ventana abierta desde la ventana 'padre', se ha de conservar la referencia a la ventana nueva devuelta por el objeto `open()`.

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
var windowHandle =
window.open('page.htm','windowName','width=600,height=320');
//-->
</SCRIPT>
```

Para evitar errores cuando referenciemos a la ventana '`opener`' desde la ventana popup, es conveniente comprobar que existe el soporte a la propiedad `opener`, y si es necesario proporcionarlo.

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
```

```
var windowHandle =
window.open('page.htm','windowName','width=600,height=320');

if (!windowHandle.opener)
    windowHandle.opener = self;
//-->
</SCRIPT>
```

A la hora de modificar el contenido o características de una ventana nueva, es conveniente esperar un tiempo después de la apertura de la ventana para permitir la carga completa y que no se produzcan errores al intentar acceder a elementos que todavía no existen.

Ejemplo:

```
<SCRIPT LANGUAGE="JavaScript">
<!--
function update() {
    windowHandle.document.open();
    windowHandle.document.write('<H1>Hello World<\/H1>');
    windowHandle.document.close();
}

var windowHandle =
window.open('page.htm','windowName','width=600,height=320');
if (!windowHandle.opener)
    windowHandle.opener = self;
setTimeout('update()',2000);
//-->
</SCRIPT>
```

Tratamiento de errores

La mayoría de lenguajes de alto nivel incluyen en sus interfaces de programación sofisticadas herramientas de depuración de código. Estas aplicaciones permiten la ejecución de los fragmentos de código erróneos por separado, y paso a paso, con la posibilidad de incorporar ventanas para visualizar los valores de las variables que intervienen en la ejecución, introducir puntos de ruptura para detener la ejecución cuando se cumple cierta condición, etc.

Desgraciadamente, JavaScript no tiene todavía un buen depurador de código, y los programadores deben echar mano del ingenio y la experiencia para localizar los errores. Este apartado pretende ser una guía de ayuda a la hora de detectar y corregir errores en los scripts.

Tipos de errores

En JavaScript pueden producirse 3 tipos de errores: errores en tiempo de carga, errores en tiempo de ejecución y errores lógicos. Vamos a ver en qué se caracteriza cada uno de ellos:

Errores en tiempo de carga.

Este tipo de errores son los que el intérprete de JavaScript encuentra en el momento en que se carga la página. En su mayoría son errores de sintaxis debidos al incumplimiento de alguna de las reglas básicas de la escritura de código en el lenguaje JavaScript. Un ejemplo típico sería la definición de una variable de tipo cadena si utilizar las comillas.

Ejemplo:

```
var cadena = ¡Hola mundo!
```

La línea anterior provocará un error en tiempo de carga. Este tipo de errores se caracterizan por la aparición en la ventana del navegador de un cuadro de advertencia de la existencia del error. En este cuadro aparecerá una descripción del error, la línea que lo contiene y su número de línea. Esta información, aunque de ayuda, puede no corresponderse con la línea real del error.

Para dificultar más nuestra labor, las distintas versiones de los navegadores muestran mensajes diferentes para el mismo error, con ventanas de aviso también diferentes.

Errores en tiempo de ejecución.

Aunque nuestra página se cargue sin ningún error, eso no quiere decir que, posteriormente no se vaya a producir un error en tiempo de ejecución del script. Este tipo de errores se deben, normalmente, a un uso inapropiado de las funciones y objetos del lenguaje. Como ejemplo típico, intentar utilizar el valor de una variable no definida produce error en tiempo de ejecución:

Ejemplo:

```
var Nombre = "Pedro";

document.write(nombre);
```

Obsérvese en el ejemplo anterior que la primera variable es `Nombre` y la segunda `nombre`, con minúscula. Como para JavaScript son dos variables distintas, la variable nombre no existe y produce un error en tiempo de ejecución.

Este tipo de errores también producen la aparición de cuadros de aviso, con el tipo de error y la línea donde podría estar el problema.

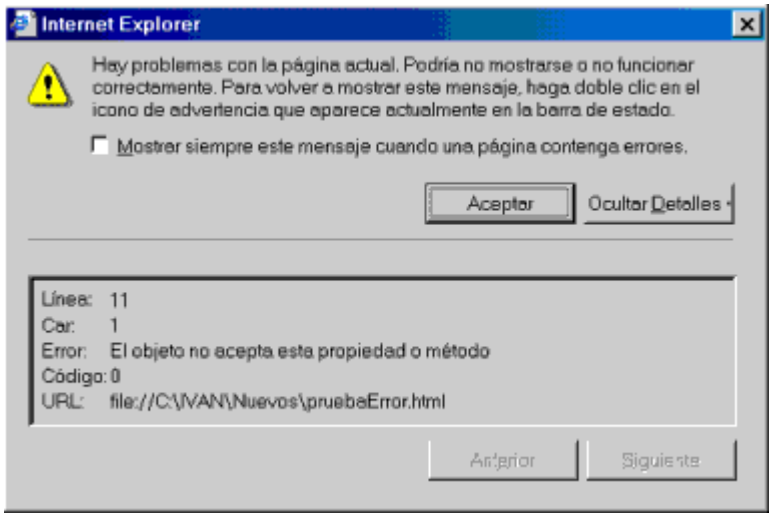
Errores lógicos.

Finalmente, podemos tener un script perfecto codificado, y sin ninguna clase de error en tiempo de ejecución, pero que cuando se desea que cambie el color de un texto a rojo, lo ponga en verde. Este tipo de problemas son debidos a errores lógicos, y suelen ser más difíciles de detectar que los anteriores.

Consolas de error

Cada navegador trata de diferente forma los errores que aparecen en el código. El Internet Explorer 4.0 abre un cuadro de aviso con cada error que aparezca en el script. Desgraciadamente, los mensajes de error que muestra son de poca ayuda, y el número de línea que indica suele estar equivocado. La versión 5.0 de este navegador mejora este comportamiento, y además permite elegir una navegación sin mensajes de error, avisándonos de que hay un error mediante un icono en la barra de estado:

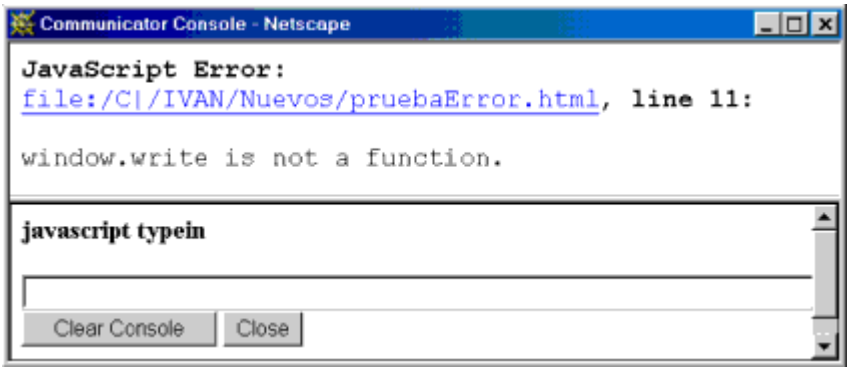
Pulsando sobre el icono, aparecerá un cuadro con información sobre los errores encontrados.



El Netscape Navigator a partir de las versiones 4.x. cuando detecta un error en la página, aparece el siguiente texto en la barra de estado:

"*JavaScript error. Type javascript: into location for details*"

Este mensaje indica que escribiendo javascript: en la barra de direcciones y pulsando intro obtendremos más información sobre el error. Al hacerlo, se abre la consola de errores JavaScript del Navigator.



En esta ventana aparecen listados los errores recientes de la página. Su contenido puede borrarse (para comenzar una nueva depuración). Si se desea, la ventana puede dejarse permanentemente abierta para mayor comodidad. También incorpora un cuadro de texto para introducir líneas de código y ejecutarlas, lo que nos permite comprobar si tienen errores.

Errores frecuentes

Vamos a describir a continuación algunos de los errores más habituales que se cometen en JavaScript.

Utilización de cadenas sin comillas.

Este error sintáctico es uno de los más frecuentes. Sobre todo cuando dentro de una cadena tenemos que usar el propio carácter de las comillas, y utilizamos para ello comillas simples y comillas dobles (JavaScript permite esta dualidad dentro de las cadenas, para facilitar la inclusión de código JavaScript en parámetros de etiquetas HTML).

Ejemplo:

```
<body onLoad="EscribeTitulo(Título de la página)">
```

En este ejemplo, se invoca la función `EscribeTitulo()` desde el evento `onLoad`. Si esta función recibe un parámetro tipo cadena, la cadena debe ir entre comillas simples, o se produce un error en tiempo de carga.

```
<body onLoad="EscribeTitulo('Título de la página')">
```

Otro error muy frecuente es olvidarse de cerrar las comillas (simples o dobles) en una sentencia que las utiliza indistintamente.

Utilización de un solo signo de igualdad en sentencias de comparación.

No es lo mismo:

```
variable1 = variable2
```

que

```
variable1 == variable2
```

Lo primero es una asignación, el contenido de la variable2 se asigna a variable1. Lo segundo es una operación de comparación, que devuelve verdadero o falso. Este error se comete habitualmente en sentencias de bifurcación.

Ejemplo:

```
if (nombre = "Pedro")
   document.write("Pedro")
```

En este ejemplo, donde se quiere hacer una comparación se está realizando una asignación.

Acceso a objetos que no existen todavía.

Otro error muy frecuente es intentar usar métodos o propiedades de objetos que todavía no existen. El procesamiento de una página HTML se produce según se va cargando. Si un script se ejecuta en la cabecera de una página, y sin que se lance su ejecución mediante una función asociada a un evento, en el momento de su ejecución pueden no existir objetos de la página como formularios, imágenes, etc. a los cuales el script puede querer acceder.

Ejemplo:

```
<html>

  <head>
    <script language="JavaScript">
      document.formulario.texto.value = "¡Hola!"
    </script>
  </head>

  <body>
    <form name="formulario">
      <input type="text" name="texto">
    </form>
  </body>

</html>
```

Este error también se produce cuando un script de un frame intenta acceder a propiedades de la página de otro frame. Puede suceder que esta página todavía no se haya cargado en el momento en que se ejecuta el script.

Uso incorrecto de la jerarquía de objetos.

Este error es muy frecuente cuando se está aprendiendo JavaScript y no se conoce la jerarquía de objetos con precisión. Se produce cuando intentamos acceder a una propiedad o método de un objeto, pero nos equivocamos en la jerarquía.

Ejemplo:

```
document.forms[0].value = "¡Hola!";
```

La propiedad `value` no existe en el objeto `forms` (que representa a un formulario), sino en los objetos cuadro de texto del formulario.

Mala colocación de las llaves en los bloques de código.

Este error consiste en la falta de alguna llave de cierre de bloque de sentencias de código, o la mala colocación de la misma. Este error, difícil de detectar, conduce a mensajes que, con frecuencia, nada tienen que ver con el error. Se produce en la definición de funciones, en la creación de sentencias de bifurcación anidadas, etc.

Ejemplo:

```
if (nombre == "Pedro" {
   document.write("¡Hola Pedro!");
else {
   alert("Hola");
}
```

Como se ve, delante del `else` falta una llave de cierre de bloque de sentencias de código, que nos llevará a un error en tiempo de carga

Uso incorrecto de los nombres de variables..

Este error tiene muchas variantes. JavaScript distingue entre mayúsculas y minúsculas, lo que hace que `miVariable` y `mivariable` sean dos variables distintas.

Por otra parte, existen una serie de palabras reservadas que no pueden utilizarse como nombres de variables. Por ejemplo, una variable en JavaScript no puede llamarse `var`, porque esta palabra reservada la usa el lenguaje para la declaración de variables.

Este problema se puede extender a la utilización incorrecta de los nombres de la jerarquía de objetos. Confusiones del tipo:

```
document.writte("¡Hola!")
```

no son infrecuentes (obsérvese cómo se ha escrito `write`).

Seguridad en JavaScript

JavaScript tiene un mecanismo de seguridad por el cual ningún script puede acceder a las propiedades de documentos que procedan de un servidor distinto. Se basa en el concepto del mismo origen, según el cual cuando se carga un documento de un determinado origen, un script cargado de otro origen (y en otra ventana o frame) no puede obtener ni establecer propiedades de objetos del primer documento.

El origen en este contexto queda definido como la parte de la URL que contiene el protocolo, el nombre de la máquina, el dominio y el puerto.

La política de seguridad por defecto, en todas las versiones de JavaScript, sigue la regla del mismo origen. No obstante, las versiones 1.1 y 1.2 de JavaScript han introducido variaciones sobre este modelo.

Marcado de datos (data tainting)

JavaScript 1.1 introduce el marcado de datos, que sólo está disponible en esta versión, habiendo sido sustituido en las versiones posteriores por el firmado de scripts. Cuando el marcado de datos está habilitado, un script de una ventana o frame puede ver las propiedades de un documento de otra ventana o frame, independientemente del origen de ambos.

Sin embargo, los desarrolladores pueden marcar (taint) los elementos de sus documentos que desean que sean privados o seguros. Si un script intenta acceder a propiedades marcadas, se mostrará un mensaje de aviso en el navegador, de modo que el usuario puede conceder o denegar el acceso a esa información. Cuando el marcado de datos está deshabilitado, se sigue la regla del mismo origen.

Algunas de las propiedades y métodos están marcadas por defecto. El autor de la página puede marcar o desmarcar los elementos de sus scripts, utilizando las funciones taint() y untaint(). En cualquier caso, para que el marcado de datos tenga efecto, debe estar habilitado, lo cual sólo puede hacerlo el usuario final, estableciendo el valor de la variable de entorno NS_ENABLE_TAINT a 1.

Firmado de scripts

La versión 1.2 de JavaScript elimina el marcado de datos, sustituyéndolo por el firmado de scripts, basado en el modelo de seguridad de Java para objetos firmados. Para firmar un script se utiliza la herramienta Netscape Signing Tool. Mediante este software se pueden asociar firmas digitales a scripts de una página HTML, a manejadores de eventos, a entidades JavaScript o a scripts en ficheros

independientes. La firma digital y un identificador del script se almacenan en ficheros .JAR.

Los scripts firmados solicitarán privilegios especiales para ganar acceso a la información restringida, utilizando las clases Java de la Java Capabilites API. Estas clases añaden funcionalidades y mejoran el control que proporciona la clase estándar Security Manager de Java.

Referencias

- Client Side JavaScript Reference.
  (http://developer.netscape.com/docs/manuals/js/client/jsguide/index.htm)
- Client Side JavaScript Guide
  (http://developer.netscape.com/docs/manuals/js/client/jsref/ClientReferenceJS13.pdf)
- NetScape JavaScript developer central
  (http://developer.netscape.com/tech/javascript/index.html)
- W3C.
  (http://www.w3c.org)
- JavaScript World
  (http://www.jsworld.com)