Unidad Didáctica

Table of Contents

1. Introducción

1.1. ¿Qué es JavaScript?

JavaScript es un lenguaje de programación que se utiliza principalmente para crear páginas web dinámicas.

Una página web dinámica es aquella que incorpora efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario.

Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

1.2. Cómo incluir JavaScript en documentos HTML

- **Incluir JavaScript en el mismo documento HTML**

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

```
<!DOCTYPE html 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=utf-8" />
                <title>Ejemplo de código JavaScript en el propio documento</title>
                <script type="text/javascript">
                        alert("Un mensaje de prueba");
                </script>

        </head>

        <body> <p>Un párrafo de texto.</p> </body>
</html>
```

  Para que la página HTML resultante sea válida, es necesario añadir el atributo `type` a la etiqueta `<script>`. Los valores que se incluyen en el atributo `type` están estandarizados y para el caso de JavaScript, el valor correcto es `text/javascript`.

  Este método se emplea cuando se define un bloque pequeño de código o cuando se quieren incluir

instrucciones específicas en un determinado documento HTML que completen las instrucciones y funciones que se incluyen por defecto en todos los documentos del sitio web.

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

- **Definir JavaScript en un archivo externo**

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

Ejemplo:

Archivo `codigo.js`

```
alert("Un mensaje de prueba");
```

Documento HTML

```
<!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=utf-8" />
     <title>Ejemplo de código JavaScript en el propio documento</title>

        <script type="text/javascript" src="/js/codigo.js"></script>

    </head>

    <body> <p>Un párrafo de texto.</p> </body>

</html>
```

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

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

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

páginas HTML que lo enlazan.

- **Incluir JavaScript en los elementos HTML**

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

```
<!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=utf-8" />
<title>Ejemplo de código JavaScript en el propio documento</title>

</head>

<body>

<p onclick="alert('Un mensaje de prueba')">Un párrafo de texto.</p>

</body>

</html>
```

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

1.3. Etiqueta noscript

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

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

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

  </noscript>

  </body>
```

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

1.4. Sintaxis

Las normas básicas que definen la sintaxis de JavaScript son las siguientes:

- No se tienen en cuenta los espacios en blanco y las nuevas líneas: como sucede con HTML, el intérprete de JavaScript ignora cualquier espacio en blanco sobrante, por lo que el código se puede ordenar de forma adecuada para entenderlo mejor (tabulando las líneas, añadiendo espacios, creando nuevas líneas, etc.)
- Se distinguen las mayúsculas y minúsculas: Si en JavaScript se intercambian mayúsculas y minúsculas el script no funciona.
- No se define el tipo de las variables: al crear una variable, no es necesario indicar el tipo de dato que almacenará. De esta forma, una misma variable puede almacenar diferentes tipos de datos durante la ejecución del script.
- No es necesario, pero sí conveniente terminar cada sentencia con el carácter de punto y coma (`;`).
- Se pueden incluir comentarios.

Ejemplo de comentario de una sola línea:

```
// a continuación se muestra un mensaje
alert("mensaje de prueba");
```

Ejemplo de comentario de varias líneas:

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

2. El primer script

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

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

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

	<head>

		<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
		 <title>El primer script</title>

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

```
                alert("Hola Mundo!");

        </script>

    </head>

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

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

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

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

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

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

3. Programación básica

3.1. Variables

- Declaración con inicialización:

  var <identificador> [= <valor>] ;

  Las variables en JavaScript se declaran mediante la palabra reservada `var`, aunque no es obligatorio.

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

La palabra reservada `var` solamente se debe indicar al definir por primera vez la variable, lo que se denomina **declarar** una variable.

Si cuando se declara una variable se le asigna también un valor, se dice que la variable ha sido **inicializada**.

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

```
var numero_1;

var numero_2;

numero_1 = 3;

numero_2 = 1;

var resultado = numero_1 + numero_2;
```

Es opcional declarar las variables. Estamos haciendo uso de una variable resultado que no ha sido declarada con var.

```
var numero_1 = 3;

var numero_2 = 1;

resultado = numero_1 + numero_2;
```

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

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

3.2. Tipos de variables

En JavaScript existen solo 3 tipos de datos básicos: Números, cadenas de texto y booleanos. El tipo de dato no se define al crear la variable sino al asignarle un valor. Además JavaScript no hace una comprobación de tipo a la hora de asignar valores a las variables por lo que una variable puede cambiar su tipo durante la ejecución del script.

Para saber el tipo de dato de nuestra variable se utiliza la función **typeof(variable)**.

La forma en la que se les asigna un valor depende del tipo de valor que se quiere almacenar (números, textos, etc.)

Numéricas

En este caso, el valor se asigna indicando directamente el número entero o decimal. Los números decimales utilizan el carácter . (punto) como separador decimal y se almacenan como un dato en coma flotante de 64bits, como los double de Java.

```
var iva = 16; // variable tipo entero

var total = 234.65; // variable tipo decimal
```

Cadenas de texto

Para asignar el valor a la variable, se encierra el valor entre comillas dobles o simples, para delimitar su comienzo y su final:

```
var mensaje = "Bienvenido a nuestro sitio web";
```

Secuencias de escape:
El mecanismo consiste en sustituir el carácter problemático por una combinación simple de caracteres.

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

De esta forma, este ejemplo que contiene comillas simples y dobles dentro del texto se puede rehacer de la siguiente forma:

```
var texto1 = 'Una frase con \'comillas simples\' dentro';
```

```
var texto2 = "Una frase con \"comillas dobles\" dentro";
```

Booleanos

Los únicos valores que pueden almacenar estas variables son `true` y `false`.

```
var clienteRegistrado = false;
var ivaIncluido = true;
```

Objetos

JavaScript es un lenguaje orientado a objetos aunque no con la potencia de otros lenguajes. Es más, para JavaScript todos los datos son objetos y como tales existen propiedades y métodos para cada tipo de dato.

Arrays

Declaración:

```
var nombre_array = [valor1, valor2, ..., valorN];
```

Ejemplo:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];
```

Acceso a un elemento:

```
nombre_array [valor]
```

Las posiciones de los elementos empiezan a contarse en el `0`

3.3. Operadores

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

3.3.1. Asignación

<identificador> = <expresion>;

var numero = 3;

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

Operador	Significado	Ejemplo
+=	Suma y asignación	dato+=8
-=	Resta y asignación	
*=	Multiplicación y asignación	
/=	División y asignación	
%=	Módulo y asignación	

3.3.2. Incremento y decremento

++ <identificador>; - - <identificador>;

<identificador>++ ; <identificador> - - ;

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

Ejemplo:

```
var numero = 5;
++numero;
alert(numero); // numero = 6
```

Es equivalente a:

```
var numero = 5;

numero = numero + 1;

alert(numero); // numero = 6
```

Ejemplo: Los operadores de incremento y decremento no solamente se pueden indicar como prefijo del nombre de la variable, sino que también es posible utilizarlos como sufijo. En este caso, su comportamiento es similar pero muy diferente ya que el incremento/decremento se realizará después de ejecutarse la instrucción donde aparece y no antes. En el siguiente ejemplo:

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

Ejemplo:

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

3.3.3. Lógicos

El resultado de cualquier operación que utilice operadores lógicos siempre es un valor lógico o *booleano*.

Operador	significado
!	Negación
&&	And
||	Or

Negación !

¿Qué sucede cuando la variable es un número o una cadena de texto? Para obtener la negación en este tipo de variables, se realiza en primer lugar su conversión a un valor booleano:

- Si la variable contiene un número, se transforma en false si vale 0 y en true para cualquier otro número (positivo o negativo, decimal o entero).

- Si la variable contiene una cadena de texto, se transforma en false si la cadena es vacía ("") y en true en cualquier otro caso.

```
var cantidad = 0;

vacio = !cantidad; // vacio = true

cantidad = 2;

vacio = !cantidad; // vacio = false
```

AND &&

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

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

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

variable1	variable2	variable1 && variable2
true	true	true
true	false	false
false	true	false
false	false	false

OR ||

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

Dato1	Dato2	dato1||dato2
False	False	False
False	True	True
True	False	True
True	True	True

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

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

3.3.4 Aritméticos

Los operadores definidos son: suma (+), resta (-), multiplicación (*) y división (/).

Operador	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto de la división entera
++	Incremento
--	decremento

Además de los cuatro operadores básicos, JavaScript define otro operador matemático que no es sencillo de entender cuando se estudia por primera vez, pero que es muy útil en algunas ocasiones.

Se trata del operador *"módulo" %*, que calcula el resto de la división entera de dos números.

El operador módulo en JavaScript se indica mediante el símbolo %, que no debe confundirse con el cálculo del porcentaje:

```
var numero1 = 10;

var numero2 = 5;

resultado = numero1 % numero2; // resultado = 0

numero1 = 9;

numero2 = 5;

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

3.3.5 Relacionales

Los operadores relacionales definidos por JavaScript son idénticos a los que definen las matemáticas: mayor que (>), menor que (<), mayor o igual (>=), menor o igual (<=), igual que (==) y distinto de (!=).

Para determinar si una letra es mayor o menor que otra, las mayúsculas se consideran menores que las minúsculas y las primeras letras del alfabeto son menores que las últimas (a es menor que b, b es menor que c, A es menor que a, etc.)

Operador	Significado
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
!=	Distinto de
==	Igual que
===	Estrictamente igual
!==	Estrictamente distinto

El resultado de todos estos operadores siempre es un valor booleano:

```
var numero1 = 3;

var numero2 = 5;

resultado = numero1 > numero2; // resultado = false

resultado = numero1 < numero2; // resultado = true
```

Cuando se usan estrictamente igual solo dará true si los datos tienen el mismo valor y además son del mismo tipo.

3.3.1 Orden de precedencia de los operadores

La precedencia de operadores determina el orden en el cual los operadores son evaluados. Los operadores con mayor precedencia son evaluados primero.

Los operadores de JavaScript tienen un orden establecido de evaluación. Este orden se denomina orden de precedencia. En la tabla siguiente se ve este orden (los de mayor prioridad van primero), teniendo presente que los operadores que aparecen juntos en una misma fila de la tabla, no tienen prioridad entre ellos, sino que se evaluarán en el orden en que hayan sido escritos en la expresión a evaluar:

Operador	Descripción
. [] ()	Acceso a campos, índice de matrices y llamada a funciones.
++ -- - ~ ! delete new typeof void	Incremento +1, decremento -1, negativo, NOT, NOT lógico borrado, crear objeto, mostrar tipo, indefinido
* / %	Multiplicación, división, módulo de división (resto)
+ - +	Suma, resta, concatenación de cadenas
<< >> >>>	Bit shifting
< <= > >=	menor que, menor que o igual, mayor que, mayor que o igual
== != === !==	Igualdad, desigualdad, identidad, no identidad
&	AND
^	XOR
|	OR
&&	AND logico
||	OR logico
?:	Condicional
=	Asignación
,	Evaluación múltiple

Los paréntesis se utilizan para alterar el orden natural de evaluación de los operadores. Una expresión con paréntesis será evaluada completa **antes** de que su valor resultante sea utilizado por el resto de instrucciones.

Para hacerse una idea más clara, veamos un ejemplo. Sea la siguiente expresión:

```
X = 78 * (96 + 3 + 45)
```

Como puedes ver, hay cinco operadores en este orden: **= * ( ) + +**

De acuerdo con su orden de precedencia, se evaluarán en este orden: **( ) * + + =**

Y esta sería la forma de operar paso a paso:

En primer lugar, se resuelve el paréntesis: `96 + 3 = 99` y a continuación `99 + 45 = 144`

Ahora se multiplica: `78 * 144 = 11232`

Por último se asigna el resultado: `X = 11232`

3.4. Estructuras de control de flujo

Estructura if ... else

Su definición formal es:

```
if(condicion)
       { <sentencia1>;
                ..
         <sentencia n>;
        }
 [else { <sentenciao1>;
                ..
         <sentenciaon>;
      } ]
```

Estructura switch

switch(valor){

case valor1:

sentencias;

break;

case valor2:

sentencias;

break;

case valor3:

sentencias;

break;

default:

sentencias;

}

El funcionamiento es el siguiente:

Se compara el valor en la clausula switch con los valores en los apartados case. Cuando hay una coincidencia se ejecutan todas las sentencias que haya hasta que encuentre la palabra break. Si no hay ninguna coincidencia, se ejecutarán las sentencias que estén despues de la clausula default.

Hay que tener cuidado con no olvidar la palabra break porque se ejecutarán todas las sentencias que haya

hasta que encuentre la palabra break o encuentre la llave de cierre.

La estructura `switch` se define mediante la palabra reservada `switch` seguida, entre paréntesis, del nombre de la variable que se va a utilizar en las comparaciones. Como es habitual, las instrucciones que forman parte del `switch` se encierran entre las llaves `{` y `}`.

Dentro del `switch` se definen todas las comparaciones que se quieren realizar sobre el valor de la variable. Cada comparación se indica mediante la palabra reservada `case` seguida del valor con el que se realiza la comparación. Si el valor de la variable utilizada por `switch` coincide con el valor indicado por `case`, se ejecutan las instrucciones definidas dentro de ese `case`.

Normalmente, después de las instrucciones de cada `case` se incluye la sentencia `break` para terminar la ejecución del `switch`, aunque no es obligatorio. Las comparaciones se realizan por orden, desde el primer `case` hasta el último, por lo que es muy importante el orden en el que se definen los `case`.

¿Qué sucede si ningún valor de la variable del `switch` coincide con los valores definidos en los `case`? En este caso, se utiliza el valor `default` para indicar las instrucciones que se ejecutan en el caso en el que ningún `case` se cumpla para la variable indicada.

Aunque `default` es opcional, las estructuras `switch` suelen incluirlo para definir al menos un valor por defecto para alguna variable o para mostrar algún mensaje por pantalla.

Estructura while

La estructura `while` permite crear bucles que se ejecutan ninguna o más veces, dependiendo de la condición indicada.

Su definición formal es:

```
while  (condicion)
  { <sentencias>
  }
```

El funcionamiento del bucle `while` se resume en: *"mientras se cumpla la condición indicada, repite indefinidamente las instrucciones incluidas dentro del bucle"*.

El siguiente ejemplo utiliza el bucle `while` para sumar todos los números menores o iguales que otro número:

```
  var resultado = 0; var numero = 100; var i = 0;

while(i <= numero)
  {
   resultado += i;
   i++;
  }

  alert(resultado);
```

Estructura do...while

El bucle de tipo `do...while` es muy similar al bucle `while`, salvo que en este caso **siempre** se ejecutan las instrucciones del bucle al menos la primera vez. Su definición formal es:

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

De esta forma, como la condición se comprueba después de cada repetición, la primera vez siempre se ejecutan las instrucciones del bucle. Es importante no olvidar que después del `while()`se debe añadir el carácter `;`(al contrario de lo que sucede con el bucle `while` simple).

Utilizando este bucle se puede calcular fácilmente el factorial de un número:

```
var resultado = 1; var numero = 5;
do {
        resultado *= numero; // resultado = resultado * numero
        numero--;
} while(numero > 0);

alert(resultado);
```

Estructura for

Su definición formal :

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

La idea del funcionamiento de un bucle `for` es la siguiente: *"mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del for. Además, después de cada repetición, actualiza el valor de las variables que se utilizan en la condición"*.

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

```
var mensaje = "Hola, estoy dentro de un bucle";

for(var  i  =  0;  i  <  5;  i++)
      { alert(mensaje);
      }
```

Estructura for...in

Una estructura de control derivada de `for` es la estructura `for...in`. Su definición exacta implica el uso de objetos, que es un elemento de programación avanzada que no se va a estudiar. Por tanto, solamente se va a presentar la estructura `for...in` adaptada a su uso en arrays. Su definición formal adaptada a los arrays es:

```
for(indice in array) { ...
  }
```

Si se quieren recorrer todos los elementos que forman un array, la estructura `for...in` es la forma más eficiente de hacerlo, como se muestra en el siguiente ejemplo:

```
var dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"];

for(i in dias) { alert(dias[i]);
  }
```

Esta estructura de control es la más adecuada para recorrer arrays (y objetos), ya que evita tener que indicar la inicialización y las condiciones del bucle `for` simple y funciona correctamente cualquiera que sea la longitud del array. De hecho, sigue funcionando igual aunque varíe el número de elementos del array.

3.5. Funciones y propiedades básicas de JavaScript

Funciones útiles para cadenas de texto

- **`length`**, calcula la longitud de una cadena de texto (el número de caracteres que la forman)

  ```
  var mensaje = "Hola Mundo";
  var numeroLetras = mensaje.length; // numeroLetras = 10
  ```

- `+` o la función **`concat()`** se emplea para concatenar varias cadenas de texto

  ```
  var mensaje1 = "Hola";

  var mensaje2 = " Mundo";

  var mensaje = mensaje1 + mensaje2; // mensaje = "Hola Mundo"

  var mensaje1 = "Hola";

  var mensaje2 = mensaje1.concat(" Mundo"); // mensaje2 = "Hola Mundo"
  ```

Las cadenas de texto también se pueden unir con variables numéricas:

```
var variable1 = "Hola ";
var variable2 = 3;
var mensaje = variable1 + variable2; // mensaje = "Hola 3"
```

- **toUpperCase**(), transforma todos los caracteres de la cadena a sus correspondientes caracteres en mayúsculas:

```
var mensaje1 = "Hola"; var mensaje2 = mensaje1.toUpperCase(); // mensaje2 = "HOLA"
```

- **toLowerCase**(), transforma todos los caracteres de la cadena a sus correspondientes caracteres en minúsculas:

```
var mensaje1 = "HolA"; var mensaje2 = mensaje1.toLowerCase(); // mensaje2 = "hola"
```

- **charAt(posicion)**, obtiene el carácter que se encuentra en la posición indicada:

```
var mensaje = "Hola";
var letra = mensaje.charAt(0); // letra = H
letra = mensaje.charAt(2); // letra = l
```

- **indexOf(caracter)**, calcula la posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si el carácter se incluye varias veces dentro de la cadena de texto, se devuelve su primera posición empezando a buscar desde la izquierda. Si la cadena no contiene el carácter, la función devuelve el valor `-1`:

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

- **lastIndexOf(caracter)**, calcula la última posición en la que se encuentra el carácter indicado dentro de la cadena de texto. Si la cadena no contiene el carácter, la función devuelve el valor `-1`:

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

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

- **substring(inicio, final)**, extrae una porción de una cadena de texto. El segundo parámetro es opcional. Si sólo se indica el parámetro `inicio`, la función devuelve la parte de la cadena original correspondiente desde esa posición hasta el final:

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(2); // porcion = "la Mundo"

porcion = mensaje.substring(5); // porcion = "Mundo"

porcion = mensaje.substring(7); // porcion = "ndo"
```

Si se indica un `inicio` negativo, se devuelve la misma cadena original:

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(-2); // porcion = "Hola Mundo"
```

Cuando se indica el inicio y el final, se devuelve la parte de la cadena original comprendida entre la posición inicial y la inmediatamente anterior a la posición final (es decir, la posición `inicio` está incluida y la posición `final`no):

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(1, 8); // porcion = "ola Mun"

porcion = mensaje.substring(3, 4); // porcion = "a"
```

Si se indica un `final` más pequeño que el `inicio`, JavaScript los considera de forma inversa, ya que automáticamente asigna el valor más pequeño al `inicio`y el más grande al `final`:

```
var mensaje = "Hola Mundo";

var porcion = mensaje.substring(5, 0); // porcion = "Hola "

porcion = mensaje.substring(0, 5); // porcion = "Hola "
```

- **`split(separador)`**, convierte una cadena de texto en un array de cadenas de texto. La función parte la cadena de texto determinando sus trozos a partir del carácter `separador` indicado:

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

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

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

Funciones útiles para arrays

- **`length`**, calcula el número de elementos de un array

```
var vocales = ["a", "e", "i", "o", "u"]; var numeroVocales = vocales.length;

// numeroVocales = 5
```

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

```
var array1 = [1, 2, 3];
```

```
array2 = array1.concat(4, 5, 6);

 // array2 = [1, 2, 3, 4, 5, 6]

array3 = array1.concat([4, 5, 6]);

// array3 = [1, 2, 3, 4, 5, 6]
```

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

```
var array = ["hola", "mundo"];

var mensaje = array.join(""); // mensaje = "holamundo"

mensaje = array.join(" "); // mensaje = "hola mundo"
```

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

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

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

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

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

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

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

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

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

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

Funciones útiles para números

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

```
var numero1 = 0;

var numero2 = 0;

alert(numero1/numero2); // se muestra el valor NaN
```

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

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

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

```
var numero1 = 10;

var numero2 = 0; alert(numero1/numero2); // se muestra el valor Infinity
```

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

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

4. Programación avanzada

4.1. Funciones

Una función es un conjunto de instrucciones que se agrupan para realizar una tarea concreta y que se pueden reutilizar fácilmente.

Definición de una función

Su definición formal es la siguiente:

```
function <nombre_funcion>(<lista_de_parámetros_formales>)
  {
```

```
<sentencias>
}
```

Ejemplo anterior, se crea una función llamada `suma_y_muestra` de la siguiente forma:

```
function suma_y_muestra(numero1,numero2)
       { resultado = numero1 + numero2;
         alert("El resultado es " + resultado);
        }
```

Llamada a la una función

Sintaxis:

```
<nombre_funcion>(<lista_de_parámetros_reales>)
```

Ejemplo:

```
// Definición de la función

function suma_y_muestra(primerNumero, segundoNumero) {
        var resultado = primerNumero + segundoNumero;
        alert("El resultado es " + resultado);
}

// Declaración de las variables
var numero1 = 3; var numero2 = 5;

// Llamada a la función
suma_y_muestra(numero1, numero2);
```

En el código anterior, se debe tener en cuenta que:

- Aunque casi siempre se utilizan variables para pasar los datos a la función, se podría haber utilizado directamente el valor de esas variables: `suma_y_muestra(3, 5);`
- El número de argumentos que se pasa a una función debería ser el mismo que el número de argumentos que ha indicado la función. No obstante, JavaScript no muestra ningún error si se pasan más o menos argumentos de los necesarios.
- El orden de los argumentos es fundamental, ya que el primer dato que se indica en la llamada, será el primer valor que espera la función; el segundo valor indicado en la llamada, es el segundo valor que espera la función y así sucesivamente.
- Se puede utilizar un número ilimitado de argumentos, aunque si su número es muy grande, se complica en exceso la llamada a la función.
- No es obligatorio que coincida el nombre de los argumentos que utiliza la función y el nombre de los argumentos que se le pasan. En el ejemplo anterior, los argumentos que se pasan son `numero1`y `numero2` y los argumentos que utiliza la función son `primerNumero` y `segundoNumero`.

Devolver un valor a través de la función

A continuación se muestra otro ejemplo de una función que calcula el precio total de un producto a partir de su precio básico:

```
// Definición de la función
function calculaPrecioTotal(precio) {
  var impuestos = 1.16;
  var gastosEnvio = 10;
  var precioTotal = ( precio * impuestos ) + gastosEnvio;

  return precioTotal;

  }

// Llamada a la función
/ El valor devuelto por la función, se guarda en una variable
resultado = calculaPrecioTotal(23.34);
```

Si no se indica el nombre de ninguna variable, JavaScript no muestra ningún error y el valor devuelto por la función simplemente se pierde y por tanto, no se utilizará en el resto del programa. En este caso, tampoco es obligatorio que el nombre de la variable devuelta por la función coincida con el nombre de la variable en la que se va a almacenar ese valor.

Si la función llega a una instrucción de tipo `return`, se devuelve el valor indicado y finaliza la ejecución de la función. Por tanto, todas las instrucciones que se incluyen después de un `return` se ignoran y por ese motivo la instrucción `return` suele ser la última de la mayoría de funciones.

4.2. Ámbito de las variables

El ámbito de una variable es la zona del programa en la que se define la variable.

JavaScript define dos ámbitos para las variables: global y local.

- El siguiente ejemplo ilustra el comportamiento de los ámbitos:

```
function creaMensaje() {
  var mensaje = “Mensaje de prueba”;

  }


  creaMensaje();

  alert(mensaje);
```

El ejemplo anterior define en primer lugar una función llamada `creaMensaje` que crea una variable llamada `mensaje`. A continuación, se ejecuta la función mediante la llamada `creaMensaje();` y seguidamente, se muestra mediante la función `alert()` el valor de una variable llamada `mensaje`.

Sin embargo, al ejecutar el código anterior no se muestra ningún mensaje por pantalla. La razón es que la variable `mensaje` se ha definido dentro de la función `creaMensaje()` y por tanto, es una **variable local** que

solamente está definida dentro de la función.

Cualquier instrucción que se encuentre dentro de la función puede hacer uso de esa variable, pero todas las instrucciones que se encuentren en otras funciones o fuera de cualquier función no tendrán definida la variable `mensaje`. De esta forma, para mostrar el mensaje en el código anterior, la función `alert()`debe llamarse desde dentro de la función `creaMensaje()`:

```
function creaMensaje() {
var mensaje = “Mensaje de prueba”;
alert(mensaje);
}


creaMensaje();
```

- Además de variables locales, también existe el concepto de variable global, que está definida en cualquier punto del programa (incluso dentro de cualquier función).

```
var mensaje = “Mensaje de prueba”;

function muestraMensaje() {
  alert(mensaje);
  }
```

El código anterior es el ejemplo inverso al mostrado anteriormente. Dentro de la función `muestraMensaje()` se quiere hacer uso de una variable llamada `mensaje` y que no ha sido definida dentro de la propia función. Sin embargo, si se ejecuta el código anterior, sí que se muestra el mensaje definido por la variable `mensaje`.

El motivo es que en el código JavaScript anterior, la variable `mensaje` se ha definido fuera de cualquier función. Este tipo de variables automáticamente son variables globales y están disponibles en cualquier punto del programa (incluso dentro de cualquier función).

De esta forma, aunque en el interior de la función no se ha definido ninguna variable llamada `mensaje`, la variable global creada anteriormente permite que la instrucción `alert()` dentro de la función muestre el mensaje correctamente.

Si una variable se declara fuera de cualquier función, automáticamente se transforma en variable global independientemente de si se define utilizando la palabra reservada `var` o no. Sin embargo, las variables definidas dentro de una función pueden ser globales o locales.

Si en el interior de una función, las variables se declaran mediante `var` se consideran locales y las variables que no se han declarado mediante `var`, se transforman automáticamente en variables globales.

Por lo tanto, se puede rehacer el código del primer ejemplo para que muestre el mensaje correctamente. Para ello, simplemente se debe definir la variable dentro de la función sin la palabra reservada `var`, para que se transforme en una variable global:

```
function creaMensaje() { mensaje = "Mensaje de prueba"; } //es vb global pq no usa var

creaMensaje();

alert(mensaje);
```

- ¿Qué sucede si una función define una variable local con el mismo nombre que una variable global que ya existe? En este caso, las variables locales prevalecen sobre las globales, pero sólo dentro de la función:

```
var mensaje = "gana la de fuera";

function muestraMensaje()
{
  var mensaje = "gana la de dentro"; //local
  alert(mensaje);

}

alert(mensaje); muestraMensaje(); alert(mensaje);
```

El código anterior muestra por pantalla los siguientes mensajes:

```
gana la de fuera gana la de dentro gana la de fuera
```

Dentro de la función, la variable local llamada `mensaje` tiene más prioridad que la variable global del mismo nombre, pero solamente dentro de la función.

- ¿Qué sucede si dentro de una función se usa una variable con el mismo nombre que otra variable global que ya existe? En este otro caso, la variable global definida dentro de la función simplemente modifica el valor de la variable global definida anteriormente:

```
var mensaje = "gana la de fuera";
function muestraMensaje()
  { mensaje = "gana la de dentro"; //uso de la vb global
    alert(mensaje);
   }

alert(mensaje); muestraMensaje(); alert(mensaje);
```

En este caso, los mensajes mostrados son:

```
gana la de fuera gana la de dentro gana la de dentro
```

La recomendación general es definir como variables locales todas las variables que sean de uso exclusivo para realizar las tareas encargadas a cada función. Las variables globales se utilizan para compartir variables entre

funciones de forma sencilla.

5. DOM

La creación del *Document Object Model* o **DOM** es una de las innovaciones que más ha influido en el desarrollo de las páginas web dinámicas y de las aplicaciones web más complejas.

DOM permite a los programadores web acceder y manipular las páginas HTML como si fueran documentos XML. De hecho, DOM se diseñó originalmente para manipular de forma sencilla los documentos XML.

A pesar de sus orígenes, DOM se ha convertido en una utilidad disponible para la mayoría de lenguajes de programación (Java, PHP, JavaScript) y cuyas únicas diferencias se encuentran en la forma de implementarlo.

5.1. Árbol de nodos

Una de las tareas habituales en la programación de aplicaciones web con JavaScript consiste en la manipulación de las páginas web. De esta forma, es habitual obtener el valor almacenado por algunos elementos (por ejemplo los elementos de un formulario), crear un elemento (párrafos, `<div>`, etc.) de forma dinámica y añadirlo a la página, aplicar una animación a un elemento (que aparezca/desaparezca, que se desplace, etc.).

Todas estas tareas habituales son muy sencillas de realizar gracias a DOM. Sin embargo, para poder utilizar las utilidades de DOM, es necesario *"transformar"* la página original. Una página HTML normal no es más que una sucesión de caracteres, por lo que es un formato muy difícil de manipular. Por ello, los navegadores web transforman automáticamente todas las páginas web en una estructura más eficiente de manipular.

Esta transformación la realizan todos los navegadores de forma automática y nos permite utilizar las herramientas de DOM de forma muy sencilla. El motivo por el que se muestra el funcionamiento de esta transformación interna es que condiciona el comportamiento de DOM y por tanto, la forma en la que se manipulan las páginas.

DOM transforma todos los documentos HTML en un conjunto de elementos llamados ***nodos***, que están interconectados y que representan los contenidos de las páginas web y las relaciones entre ellos. Por su aspecto, la unión de todos los nodos se llama *"árbol de nodos"*.

La siguiente página HTML sencilla:

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

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

	<head>

		<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
		<title>Página sencilla</title>

	</head>
```

```
	<body>
		<p>Esta página es <strong>muy sencilla</strong></p>
	</body>
</html>
```

Se transforma en el siguiente árbol de nodos:

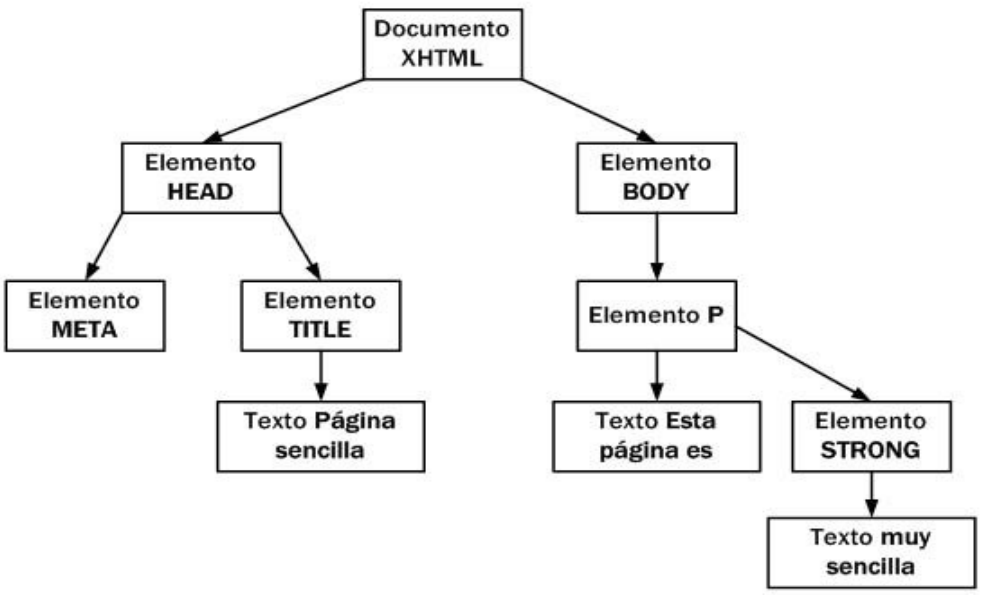

Figura . Árbol de nodos generado automáticamente por DOM a partir del código XHTML de la página

En el esquema anterior, cada rectángulo representa un nodo DOM y las flechas indican las relaciones entre nodos. Dentro de cada nodo, se ha incluido su tipo (que se verá más adelante) y su contenido.

La **raíz del árbol** de nodos de cualquier página HTML siempre es la misma: un nodo de tipo especial denominado ***"Documento"***.

A partir de ese nodo raíz, cada etiqueta HTML se transforma en un nodo de tipo *"Elemento"*. La conversión de etiquetas en nodos se realiza de forma jerárquica. De esta forma, del nodo raíz solamente pueden derivar los nodos `HEAD` y `BODY`. A partir de esta derivación inicial, cada etiqueta HTML se transforma en un nodo que deriva del nodo correspondiente a su *"etiqueta padre"*.

La transformación de las etiquetas HTML habituales genera dos nodos: el primero es el nodo de tipo *"Elemento"* (correspondiente a la propia etiqueta HTML) y el segundo es un nodo de tipo *"Texto"* que contiene el texto encerrado por esa etiqueta HTML.

Así, la siguiente etiqueta HTML:

```
<title>Página sencilla</title>
```

Genera los siguientes dos nodos:

Figura Nodos generados automáticamente por DOM para una etiqueta HTML sencilla

De la misma forma, la siguiente etiqueta HTML:

```
<p>Esta página es <strong>muy sencilla</strong></p>
```

Genera los siguientes nodos:

- Nodo de tipo *"Elemento"* correspondiente a la etiqueta <p>.
- Nodo de tipo *"Texto"* con el contenido textual de la etiqueta <p>.
- Como el contenido de <p> incluye en su interior otra etiqueta HTML, la etiqueta interior se transforma en un nodo de tipo *"Elemento"* que representa la etiqueta y que deriva del nodo anterior.
- El contenido de la etiqueta genera a su vez otro nodo de tipo *"Texto"* que deriva del nodo generado por .

Figura. Nodos generados automáticamente por DOM para una etiqueta HTML con otras etiquetas HTML en su interior

La transformación automática de la página en un árbol de nodos siempre sigue las mismas reglas:

- Las etiquetas HTML se transforman en dos nodos: el primero es la propia etiqueta y el segundo nodo es hijo del primero y consiste en el contenido textual de la etiqueta.
- Si una etiqueta HTML se encuentra dentro de otra, se sigue el mismo procedimiento anterior, pero los nodos generados serán nodos hijo de su etiqueta padre.

Como se puede suponer, las páginas HTML habituales producen árboles con miles de nodos. Aun así, el proceso de transformación es rápido y automático, siendo las funciones proporcionadas por DOM (que se verán más adelante) las únicas que permiten acceder a cualquier nodo de la página de forma sencilla e inmediata.

5.2. Tipos de nodos

La especificación completa de DOM define 12 tipos de nodos, aunque las páginas HTML habituales se pueden manipular manejando solamente cuatro o cinco tipos de nodos:

- **`Document`**, nodo raíz del que derivan todos los demás nodos del árbol.
- **`Element`**, representa cada una de las etiquetas HTML. Se trata del único nodo que puede contener atributos y el único del que pueden derivar otros nodos.
- **`Attr`**, se define un nodo de este tipo para representar cada uno de los atributos de las etiquetas HTML, es decir, uno por cada par `atributo=valor`.
- **`Text`**, nodo que contiene el texto encerrado por una etiqueta HTML.
- **`Comment`**, representa los comentarios incluidos en la página HTML.

Los otros tipos de nodos existentes que no se van a considerar son `DocumentType`, `CDataSection`, `DocumentFragment`, `Entity`, `EntityReference`, `ProcessingInstruction` y `Notation`.

5.3. Acceso directo a los nodos

Una vez construido automáticamente el árbol completo de nodos DOM, ya es posible utilizar las funciones DOM para acceder de forma directa a cualquier nodo del árbol. Como acceder a un nodo del árbol es equivalente a acceder a *"un trozo"* de la página, una vez construido el árbol, ya es posible manipular de forma sencilla la página: acceder al valor de un elemento, establecer el valor de un elemento, mover un elemento de la página, crear y añadir nuevos elementos, etc.

DOM proporciona dos métodos alternativos para acceder a un nodo específico:

- acceso a través de sus nodos padre y
- acceso directo.

Las funciones que proporciona DOM para acceder a un nodo a través de sus nodos padre consisten en acceder al nodo raíz de la página y después a sus nodos hijos y a los nodos hijos de esos hijos y así sucesivamente hasta el último nodo de la rama terminada por el nodo buscado. Sin embargo, cuando se quiere acceder a un nodo específico, es mucho más rápido acceder directamente a ese nodo y no llegar hasta él descendiendo a través de todos sus nodos padre.

Por ese motivo, no se van a presentar las funciones necesarias para el acceso jerárquico de nodos y se

muestran solamente las que permiten acceder de forma directa a los nodos.

Por último, es importante recordar que el acceso a los nodos, su modificación y su eliminación solamente es posible cuando el árbol DOM ha sido construido completamente, es decir, después de que la página HTML se cargue por completo. Más adelante se verá cómo asegurar que un código JavaScript solamente se ejecute cuando el navegador ha cargado entera la página HTML.

Todos los elementos de una página se consideran **objetos**.
Un objeto es un elemento de programación con dos características principales:

- propiedades, que son los datos que contiene el elemento y que se pueden obtener o modificar. Para acceder a la propiedad se utiliza la sintaxis: objeto.propiedad.
  Podemos asignarle un valor (objeto.propiedad=9) o utilizar el valor de la propiedad (dato=objeto.propiedad)

- métodos, que son las acciones que se pueden hacer con esos objetos. Para utilizar los eventos de un objeto, se utiliza la sintaxis objeto.metodo(). Estos métodos suelen devolver un valor que se puede utilizar luego (dato=objeto.metodo()).

Todos los elementos son propiedad del objeto **document**. El objeto document representa a la página web.

5.3.1. getElementsByTagName()

La función `getElementsByTagName(nombreEtiqueta)` *obtiene todos los elementos de la página HTML cuya etiqueta sea igual que el parámetro que se le pasa a la función*.

El siguiente ejemplo muestra cómo obtener todos los párrafos de una página HTML:

```
var parrafos = document.getElementsByTagName("p");
```

El valor que se indica delante del nombre de la función (en este caso, `document`) es el nodo a partir del cual se realiza la búsqueda de los elementos. En este caso, como se quieren obtener todos los párrafos de la página, se utiliza el valor `document` como punto de partida de la búsqueda.

El valor que devuelve la función es un array con todos los nodos que cumplen la condición de que su etiqueta coincide con el parámetro proporcionado. *El valor devuelto es un array de nodos DOM*, no un array de cadenas de texto o un array de objetos normales. Por lo tanto, se debe procesar cada valor del array de la forma que se muestra en las siguientes secciones.

De este modo, se puede obtener el primer párrafo de la página de la siguiente manera:

```
var primerParrafo = parrafos[0];
```

De la misma forma, se podrían recorrer todos los párrafos de la página con el siguiente código:

```
for(var i=0; i<parrafos.length; i++)

        alert( parrafos[i]);
```

La función `getElementsByTagName()` se puede aplicar de forma recursiva sobre cada uno de los nodos devueltos por la función.

En el siguiente ejemplo, se obtienen todos los enlaces del primer párrafo de la página:

```
var parrafos = document.getElementsByTagName("p");

var primerParrafo = parrafos[0];

var enlaces = primerParrafo.getElementsByTagName("a");
```

5.3.2. getElementsByName()

La función `getElementsByName()` es similar a la anterior, pero en este caso se buscan los *elementos cuyo atributo `name` sea igual al parámetro proporcionado*. En el siguiente ejemplo, se obtiene directamente el único párrafo con el nombre indicado:

```
var parrafoEspecial = document.getElementsByName("especial");

<p name="prueba">...</p> <p name="especial">...</p>
<p>...</p>
```

Normalmente el atributo `name` es único para los elementos HTML que lo definen, por lo que es un método muy práctico para acceder directamente al nodo deseado. En el caso de los elementos HTML *radiobutton*, el atributo `name` es común a todos los *radiobutton* que están relacionados, por lo que la función devuelve una colección de elementos.

5.3.3. getElementById()

La función `getElementById()` es la más utilizada cuando se desarrollan aplicaciones web dinámicas. Se trata de la función preferida para acceder directamente a un nodo y poder leer o modificar sus propiedades.

La función `getElementById()` *devuelve el elemento HTML cuyo atributo `id` coincide con el parámetro indicado en la función.* Como el atributo `id` debe ser único para cada elemento de una misma página, la

función devuelve únicamente el nodo deseado.

```
var cabecera = document.getElementById("cabecera");

<div id="cabecera">
  <a href="/" id="logo">...</a> </div>
```

La función `getElementById()` es tan importante y tan utilizada en todas las aplicaciones web, que casi todos los ejemplos y ejercicios que siguen la utilizan constantemente.

5.3.4. GetElementsByClassName()

Funciona exactamente igual que getElementsByName pero utilizando el nombre de la clase.

5.4. Creación y eliminación de nodos

Acceder a los nodos y a sus propiedades es sólo una parte de las manipulaciones habituales en las páginas. Las otras operaciones habituales son las de crear y eliminar nodos del árbol DOM, es decir, crear y eliminar *"trozos"* de la página web.

5.4.1. Creación de elementos HTML simples

Como se ha visto, un elemento HTML sencillo, como por ejemplo un párrafo, genera dos nodos: el primer nodo es de tipo `Element` y representa la etiqueta `<p>` y el segundo nodo es de tipo `Text` y representa el contenido textual de la etiqueta `<p>`.

Por este motivo, crear y añadir a la página un nuevo elemento HTML sencillo consta de cuatro pasos diferentes:

1. Creación de un nodo de tipo `Element` que represente al elemento.
2. Creación de un nodo de tipo `Text` que represente el contenido del elemento.
3. Añadir el nodo `Text` como nodo hijo del nodo `Element`.
4. Añadir el nodo `Element` a la página, en forma de nodo hijo del nodo correspondiente al lugar en el que se quiere insertar el elemento.

De este modo, si se quiere añadir un párrafo simple al final de una página HTML, es necesario incluir el siguiente código JavaScript:

```
// Crear nodo de tipo Element
var parrafo = document.createElement("p");

// Crear nodo de tipo Text
var contenido = document.createTextNode("Hola Mundo!");

// Añadir el nodo Text como hijo del nodo Element
parrafo.appendChild(contenido);
```

```
// Añadir el nodo Element como hijo de la pagina
document.body.appendChild(parrafo);
```

El proceso de creación de nuevos nodos puede llegar a ser tedioso, ya que implica la utilización de tres funciones DOM:

- **`createElement(etiqueta)`**: crea un nodo de tipo `Element`que representa al elemento HTML cuya etiqueta se pasa como parámetro.
- **`createTextNode(contenido)`**: crea un nodo de tipo `Text` que almacena el contenido textual de los elementos HTML.
- **`nodoPadre.appendChild(nodoHijo)`**: añade un nodo como hijo de otro nodo. Se debe utilizar al menos dos veces con los nodos habituales: en primer lugar se añade el nodo `Text`como hijo del nodo `Element` y a continuación se añade el nodo `Element`como hijo de algún nodo de la página.

5.4.2. Eliminación de nodos

Es necesario utilizar la función `removeChild()`:

```
var parrafo = document.getElementById("provisional");

parrafo.parentNode.removeChild(parrafo);

<p id="provisional">...</p>
```

La función `removeChild()` requiere como parámetro el nodo que se va a eliminar. Además, esta función debe ser invocada desde el elemento padre de ese nodo que se quiere eliminar. La forma más segura y rápida de acceder al nodo padre de un elemento es mediante la propiedad `nodoHijo.parentNode`.

Así, para eliminar un nodo de una página HTML se invoca a la función `removeChild()` desde el valor `parentNode` del nodo que se quiere eliminar. Cuando se elimina un nodo, también se eliminan automáticamente todos los nodos hijos que tenga, por lo que no es necesario borrar manualmente cada nodo hijo.

5.5. Acceso directo a los atributos HTML

5.5.1 Cambiar el valor de un atributo directamente

Para cambiar el valor de un atributo se puede utilizar la siguiente sintaxis:

```
document.getElementById(id).attribute=valor
```

```
<html>
	<body>
		<img id="image" src="smiley.gif">
		<script>
			document.getElementById("image").src="landscape.jpg";
		</script>
	</body>
</html>
```

Sea el trozo de HTML:

```
<a id="enlace" href="http://www...com">Enlace</a>
```

El siguiente ejemplo obtiene de forma directa la dirección a la que enlaza el enlace:

```
var enlace = document.getElementById("enlace");

alert(enlace.href); // muestra http://www...com
```

5.5.2 Cambiar un atributo usando setAttribute

Podemos cambiar el valor de los atributos mediante el método setAttribute(nombre_atributo,valor)

```
<html>
	<body>
		<img id="image" src="smiley.gif">
		<script>
			dato=document.getElementById("image");
			dato.setAttribute("src","landscape.jpg");
		</script>
	</body>
</html>
```

5.5.3 Eliminar un atributo

Podemos eliminar un atributo de un elemento usando el método removeAttribute(nombre_atributo)

```
<html>
	<body>
		<img id="image" src="smiley.gif" width=”200”>
		<script>
			dato=document.getElementById("image");
			dato.removeAttribute(“width”);
		</script>
	</body>
</html>
```

5.5.4 Obtener el valor de un atributo

Para obtener el valor de un atributo,podemos leerlo directamente o utilizar el método getAttribute(“nombre_atributo).

```
<html>
<body>
<img id="image" src="smiley.gif">
<script>
dato=document.getElementById("image").src;
elem=document.getElementById("image"); //otra forma
dato=elem.getAttribute(“src”);
</script>
</body>
</html>
```

5.5.5 Cambiar estilos CSS

Para cambiar elementos CSS se puede utilizar la sintaxis:

document.getElementById(id).style.property=nuevo_dato;

```
<html>
	<body>
		<p id="p2">Hello World!</p>
		<script>
			document.getElementById("p2").style.color="blue";
		</script>
```

```
            <p >The paragraph above was changed by a script.</p>
        </body>
    </html>
```

La lista de elementos que podemos cambiar es enorme e incluyen todas las propiedades de CSS

5.5.4 Obtener las propiedades CSS de los elementos

JavaScript solo puede obtener los valores que previamente haya establecido. Es decir, no podemos ver directamente los valores de estilo de un elemento de la página.

Existe una forma y es utilizando el método getComputedStyle() del objeto window:

var style = window.getComputedStyle(elemento);

Esto nos devolverá un objeto style que incluye todos los atributos del elemento.

x=document.getElementById("caja");

estilo=window.getComputedStyle(x);

estilo.width

Las propiedades CSS no son tan fáciles de obtener como los atributos HTML. Para obtener el valor de cualquier propiedad CSS del nodo, se debe utilizar el atributo `style`.

Sea: `<img id="imagen" style="margin:0; border:0;" src="logo.png" />`

El siguiente ejemplo obtiene el valor de la propiedad `margin` de la imagen:

```
var imagen = document.getElementById("imagen");

alert(imagen.style.margin);
```

Aunque el funcionamiento es homogéneo entre distintos navegadores, los resultados no son exactamente iguales.

Si el nombre de una propiedad CSS es compuesto, se accede a su valor modificando ligeramente su nombre:

```
var parrafo = document.getElementById("parrafo");

alert(parrafo.style.fontWeight); // muestra "bold"
```

```
<p id="parrafo" style="font-weight: bold;">...</p>
```

La transformación del nombre de las propiedades CSS compuestas consiste en eliminar todos los guiones medios (`-`) y escribir en mayúscula la letra siguiente a cada guión medio. A continuación se muestran algunos ejemplos:

- `font-weight` se transforma en `fontWeight`
- `line-height` se transforma en `lineHeight`
- `border-top-style` se transforma en `borderTopStyle`
- `list-style-image` se transforma en `listStyleImage`

El único atributo HTML que no tiene el mismo nombre en HTML y en las propiedades DOM es el atributo `class`. Como la palabra `class` está reservada por JavaScript, no es posible utilizarla para acceder al atributo `class` del elemento HTML. En su lugar, DOM utiliza el nombre `className` para acceder al atributo `class` de HTML:

```
var parrafo = document.getElementById("parrafo");

alert(parrafo.class); // muestra "undefined"

alert(parrafo.className); // muestra "normal"

<p id="parrafo" class="normal">...</p>
```

5.5. Otras acciones

5.5.1. La propiedad innerHTML

Es la forma más fácil de obtener o reemplazar el contenido de un elemento.

```
<html>
	<body>
		<p id="intro">Hello World!</p>
		<script>
			var txt=document.getElementById("intro").innerHTML;
			document.write(txt);
		</script>
	</body>
</html>
```

Podemos utilizar el mismo método para cambiar el contenido de una etiqueta HTML.

```
	<html>
		<head>
			<title>EJEMPLO Cambio</title>
			<meta http-equiv="Content-Type" content="text/html;charset=UTF-8">
			<script type="text/javascript">
				function escribir(){
					document.getElementById("p1").innerHTML="Buenos
días";
				}
			</script>
		</head>
	<body>
	<p id="p1">Cambiamos texto de la página</p>
	<p><a href="#" onclick="escribir()" >Pincha para cambiar</a></p>
</body>
</html>
```

6. Eventos

JavaScript permite asignar una función a cada uno de los eventos. De esta forma, cuando se produce cualquier evento, JavaScript ejecuta su función asociada. Este tipo de funciones se denominan *"event handlers"* en inglés y suelen traducirse por *"manejadores de eventos"*.

6.1. Modelos de eventos

Existen hasta tres modelos diferentes para manejar los eventos dependiendo del navegador en el que se ejecute la aplicación.

6.1.1. Modelo básico de eventos

Este modelo simple de eventos se introdujo para la versión 4 del estándar HTML y se considera parte del nivel más básico de DOM. Aunque sus características son limitadas, es el único modelo que es compatible en todos los navegadores y por tanto, el único que permite crear aplicaciones que funcionan de la misma manera en todos los navegadores.

6.1.2. Modelo de eventos estándar

Las versiones más avanzadas del estándar DOM (DOM nivel 2) definen un modelo de eventos completamente nuevo y mucho más poderoso que el original. Todos los navegadores modernos lo incluyen, salvo Internet Explorer.

6.1.3. Modelo de eventos de Internet Explorer

Internet Explorer utiliza su propio modelo de eventos, que es similar pero incompatible con el modelo estándar. Se utilizó por primera vez en Internet Explorer 4 y Microsoft decidió seguir utilizándolo en el resto de versiones, a pesar de que la empresa había participado en la creación del estándar de DOM que define el modelo de eventos estándar.

6.2. Modelo básico de eventos

6.2.1. Tipos de eventos

En este modelo, cada elemento o etiqueta HTML define su propia lista de posibles eventos que se le pueden asignar. Un mismo tipo de evento (por ejemplo, pinchar el botón izquierdo del ratón) puede estar definido para varios elementos HTML diferentes y un mismo elemento HTML puede tener asociados varios eventos diferentes.

El nombre de cada evento se construye mediante el prefijo `on`, seguido del nombre en inglés de la acción

asociada al evento. Así, el evento de pinchar un elemento con el ratón se denomina `onclick` y el evento asociado a la acción de mover el ratón se denomina `onmousemove`.

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

onmouseout	El ratón *"sale"* del elemento (pasa por encima de otro elemento)	Todos los elementos
onmouseover	El ratón *"entra"* en el elemento (pasa por encima del elemento)	Todos los elementos
onmouseup	Soltar el botón que estaba pulsado en el ratón	Todos los elementos
onreset	Inicializar el formulario (borrar todos sus datos)	<form>
onresize	Se ha modificado el tamaño de la ventana del navegador	<body>
onselect	Seleccionar un texto	<input>, <textarea>
onsubmit	Enviar el formulario	<form>
onunload	Se abandona la página (por ejemplo al cerrar el navegador)	<body>

Los eventos más utilizados en las aplicaciones web tradicionales son `onload` para esperar a que se cargue la página por completo, los eventos `onclick`, `onmouseover`, `onmouseout` para controlar el ratón y `onsubmit` para controlar el envío de los formularios.

Evento	**Descripción**	**Elementos para los que está definido**
onblur	Deseleccionar el elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onchange	Deseleccionar un elemento que se ha modificado	<input>, <select>, <textarea>
onclick	Pinchar y soltar el ratón	Todos los elementos
ondblclick	Pinchar dos veces seguidas con el ratón	Todos los elementos
onfocus	Seleccionar un elemento	<button>, <input>, <label>, <select>, <textarea>, <body>
onkeydown	Pulsar una tecla (sin soltar)	Elementos de formulario y <body>
onkeypress	Pulsar una tecla	Elementos de formulario y <body>
onkeyup	Soltar una tecla pulsada	Elementos de formulario y <body>
onload	La página se ha cargado completamente	<body>
onmousedown	Pulsar (sin soltar) un botón del ratón	Todos los elementos
onmousemove	Mover el ratón	Todos los elementos

Las acciones típicas que realiza un usuario en una página web pueden dar lugar a una sucesión de eventos. Al pulsar por ejemplo sobre un botón de tipo `<input type="submit">` se desencadenan los eventos `onmousedown`, `onclick`, `onmouseup` y `onsubmit` de forma consecutiva.

6.2.2. Manejadores de eventos

Para que los eventos resulten útiles, se deben asociar funciones o código JavaScript a cada evento. De esta forma, cuando se produce un evento se ejecuta el código indicado, por lo que la aplicación puede *responder* ante cualquier evento que se produzca durante su ejecución.

Las funciones o código JavaScript que se definen para cada evento se denominan *"manejador de eventos"* y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores:

- Manejadores como atributos de los elementos HTML.
- Manejadores como funciones JavaScript externas.
- Manejadores *"semánticos"*.

6.2.2.1. Manejadores de eventos como atributos HTML

El código se incluye en un atributo del propio elemento HTML.

- En el siguiente ejemplo, se quiere mostrar un mensaje cuando el usuario pinche con el ratón sobre un botón:

```
<input type="button" value="Pinchame y verás" onclick="alert('Gracias por pinchar');" />
```

En este método, se definen atributos HTML con el mismo nombre que los eventos que se quieren manejar.

- En este otro ejemplo, cuando el usuario pincha sobre el elemento `<div>` se muestra un mensaje y cuando el usuario pasa el ratón por encima del elemento, se muestra otro mensaje:

```
<div onclick="alert('Has pinchado con el ratón');" onmouseover="alert('Acabas de pasar el ratón por encima');">

Puedes pinchar sobre este elemento o simplemente pasar el ratón por encima

</div>
```

- Este otro ejemplo incluye una de las instrucciones más utilizadas en las aplicaciones JavaScript más antiguas:

```
<body onload="alert('La página se ha cargado completamente');"> ...
</body>
```

El mensaje anterior se muestra después de que la página se haya cargado completamente, es decir,

después de que se haya descargado su código HTML, sus imágenes y cualquier otro objeto incluido en la página.

El evento `onload` es uno de los más utilizados ya que las funciones que permiten acceder y manipular los nodos del árbol DOM solamente están disponibles cuando la página se ha cargado completamente.

6.2.2.2. Manejadores de eventos y variable this

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

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

Elemento `<div>`original:

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

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

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

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

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

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

6.2.2.3. Manejadores de eventos como funciones externas

La definición de los manejadores de eventos en los atributos HTML es el método más sencillo pero menos aconsejable de tratar con los eventos en JavaScript. El principal inconveniente es que se complica en exceso en cuanto se añaden algunas pocas instrucciones, por lo que solamente es recomendable para los casos más sencillos.

Si se realizan aplicaciones complejas, como por ejemplo la validación de un formulario, es aconsejable agrupar todo el código JavaScript en una función externa y llamar a esta función desde el elemento HTML.

- Siguiendo con el ejemplo anterior que muestra un mensaje al pinchar sobre un botón:

```
<input type="button" value="Pinchame y verás" onclick="alert('Gracias por pinchar');" />
```

- Utilizando funciones externas se puede transformar en:

```
function muestraMensaje() { alert('Gracias por pinchar'); }

<input type="button" value="Pinchame y verás" onclick="muestraMensaje()" />
```

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

```
  function resalta(elemento) {
switch(elemento.style.borderColor) {
 case '#c0c0c0':
  elemento.style.borderColor = 'black';
    break;
  case 'black':
    elemento.style.borderColor = 'silver'; break; }
  }

  <div style="width:150px; height:60px; border:thin solid silver"
  onmouseover="resalta(this)" onmouseout="resalta(this)"> Sección de
  contenidos... </div>
```

En el ejemplo anterior, la función externa es llamada con el parámetro `this`, que dentro de la función se denomina `elemento`. La complejidad del ejemplo se produce sobre todo por la forma en la que los distintos navegadores almacenan el valor de la propiedad `borderColor`.

6.2.2.4. Manejadores de eventos semánticos

Los métodos que se han visto para añadir manejadores de eventos (como atributos HTML y como funciones externas) tienen un grave inconveniente: *"ensucian"* el código HTML de la página.

Afortunadamente, existe un método alternativo para definir los manejadores de eventos de JavaScript. Esta técnica es una evolución del método de las funciones externas, ya que se basa en utilizar las propiedades DOM de los elementos HTML para asignar todas las funciones externas que actúan de manejadores de eventos.

Así, el siguiente ejemplo:

```
<input id="pinchable" type="button" value="Pinchame y verás"

onclick="alert('Gracias por pinchar');" />
```

Se puede transformar en:

```
// Función externa
function muestraMensaje() { alert('Gracias por pinchar'); }

// Asignar la función externa al elemento
document.getElementById("pinchable").onclick = muestraMensaje;

// Elemento HTML
<input id="pinchable" type="button" value="Pinchame y verás" />
```

La técnica de los manejadores semánticos consiste en:

1. *Asignar un identificador único al elemento HTML mediante el atributo `id`.*
2. *Crear una función de JavaScript encargada de manejar el evento.*
3. *Asignar la función externa al evento correspondiente en el elemento deseado.*

El último paso es la clave de esta técnica. En primer lugar, se obtiene el elemento al que se desea asociar la función externa:

```
document.getElementById("pinchable");
```

A continuación, se utiliza una propiedad del elemento con el mismo nombre que el evento que se quiere manejar. En este caso, la propiedad es `onclick`:

```
document.getElementById("pinchable").onclick = ...
```

Por último, se asigna la función externa mediante su nombre sin paréntesis. Lo más importante (y la causa más común de errores) es indicar solamente el nombre de la función, es decir, prescindir de los paréntesis al asignar la función:

```
document.getElementById("pinchable").onclick = muestraMensaje;
```

Si se añaden los paréntesis después del nombre de la función, en realidad se está ejecutando la función y guardando el valor devuelto por la función en la propiedad `onclick` de elemento.

```
// Asignar una función externa a un evento de un elemento
document.getElementById("pinchable").onclick = muestraMensaje;

// Ejecutar una función y guardar su resultado en una propiedad de un elemento
document.getElementById("pinchable").onclick = muestraMensaje();
```

La gran ventaja de este método es que el código HTML resultante es muy *"limpio"*, ya que no se mezcla con el código JavaScript. Además, dentro de las funciones externas asignadas sí que se puede utilizar la variable `this` para referirse al elemento que provoca el evento.

El único inconveniente de este método es que la página se debe cargar completamente antes de que se puedan utilizar las funciones DOM que asignan los manejadores a los elementos HTML. Una de las formas más sencillas de asegurar que cierto código se va a ejecutar después de que la página se cargue por completo es utilizar el evento `onload`:

```
window.onload = function() {
 document.getElementById("pinchable").onclick = muestraMensaje; }
```

La técnica anterior utiliza el concepto de *funciones anónimas*, que no se va a estudiar, pero que permite crear un código compacto y muy sencillo. Para asegurarse que un código JavaScript va a ejecutarse después de que la página se haya cargado completamente, sólo es necesario incluir esas instrucciones entre los símbolos `{` y `}`:

```
window.onload = function() { ...
}
```

En el siguiente ejemplo, se añaden eventos a los elementos de tipo `input=text` de un formulario complejo:

```
function resalta() {
  // Código JavaScript
}

window.onload = function()

{ var formulario = document.getElementById("formulario");

  var camposInput = formulario.getElementsByTagName("input");

   for(var i=0; i<camposInput.length; i++)

     {

            if(camposInput[i].type == "text")
                  { camposInput[i].onclick = resalta; }

      }

 }
```