JavaScript

Universitat Politecnica de Catalunya (UPC)

Outline

Outline

JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

¿Qué es JavaScript?

- **JavaScript** es un lenguaje de programación interpretado:
  - No es necesario compilar los programas para ejecutarlos porque es un lenguaje de programación interpretado.
  - Los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.
- Proviene de un dialecto del estándar ECMAScript.
- Java y JavaScript no están relacionados y tienen semánticas y propósitos diferentes.
- Se define como orientado a objetos, basado en prototipos, imperativo, débilmente tipado y dinámico.
- Usos:
  - Se utiliza principalmente en el lado del cliente en servicios Web, implementado como parte de un navegador web permitiendo crear interacción con el usuario y páginas web dinámicas.
  - Aunque actualmente es posible ejecutar JavaScript en el propio servidor mediante NodeJS.

Glosario Básico

- **Script**: Cada uno de los programas, aplicaciones o trozos de código creados con el lenguaje de programación JavaScript.
- **Sentencia**: Cada una de las instrucciones que forman un script.
- **Palabras reservadas**: Son las palabras (en inglés) que se utilizan para construir las sentencias de JavaScript.
  - Las palabras reservadas no pueden ser utilizadas en el código.
  - Ejemplos: break, case, catch, continue, default, delete, do, if, etc.

Sintaxis

Normas básicas de la sintaxis JS:

1. No se tienen en cuenta los espacios en blanco ni los saltos de línea.
2. Se distinguen las mayúsculas y minúsculas.
3. No se define el tipo de las variables (no es necesario indicar el tipo de dato que almacenará la variable).
4. No es obligatorio terminar cada sentencia Javascript con punto y coma **(;)** (aunque se considera una buena prática utilizar el punto y coma para finalizar las sentencias Javascript).
5. Se pueden incluir comentarios de dos tipos:
   - Los de una sola línea: se definen añadiendo dos barras oblicuas al principio del comentario *//comentario*.
   - Los que ocupan varias líneas: se definen encerrando el texto del comentario entre los símbolos */* ..... */*

¿Cómo incluir JavaScript en documentos XHTML?

- La integración de JavaScript y XHTML es muy flexible.
- Existen esencialmente tres formas para incluir código JavaScript en las páginas XHTML:

1. Incluir JavaScript en el mismo documento XHTML.
2. Definir JavaScript en un archivo externo y enlazarlo en el XHTML.
3. Incluir JavaScript en los elementos del XHTML.

JS en documentos XHTML I

- 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>**):

```
1  <html xmlns="http://www.w3.org/1999/xhtml">
2  <head>
3  <title>Ejemplo de codigo JavaScript en el propio documento</title>
4  <script type="text/javascript">
5    alert("Un mensaje de prueba");
6  </script>
7  </head>
8  <body>
9  <p>Un texto.</p>
10 </body>
11 </html>
```

JS en documentos XHTML II

- Para que la página XHTML resultante sea válida, es necesario añadir el atributo type a la etiqueta <script>.
- Técnicamente, el atributo type se corresponde con el tipo “MIME“ (estándar para identificar los diferentes tipos de contenidos).
- El tipo MIME para JavaScript es **text/javascript**.
- Este método se utiliza habitualmente cuando:
  - Sólo es necesario definir pequeño bloque de código.
  - Se quieren incluir instrucciones específicas para un determinado documento XHTML (no reutilizables).
  - El principal inconveniente es que si se quiere hacer una modificación en el bloque de código, es necesario modificar cada páginas que incluya el código JavaScript.

JS en un Archivo I

- Las instrucciones JavaScript se pueden incluir en un archivo externo mediante la etiqueta **<script>**.
- Ejemplo para enlazar un archivo en un documento XHTML:

```
1  <html xmlns="http://www.w3.org/1999/xhtml">
2  <head>
3  <title>Ejemplo de codigo JavaScript en el propio documento</title>
4  <script type="text/javascript" src="/js/codigo.js"></script>
5  </head>
6  <body>
7  <p>Un texto.</p>
8  </body>
9  </html>
```

- El archivo **codigo.js** podría contener la siguiente instrucción:

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

- Además del atributo type, este método requiere definir el atributo **src** con la URL del archivo.

JS en un Archivo II

- Cada etiqueta <script> solamente puede enlazar un único archivo.
- En una misma página se pueden incluir tantas etiquetas <script> como sean necesarias.
- Los archivos JavaScript son documentos normales (se pueden crear con cualquier editor de texto o con entornos de desarrollo) con la extensión **.js**.
- La principales ventajas de enlazar archivos JavaScript son que:
  1. Se simplifica el código XHTML de la página.
  2. Se puede reutilizar el mismo código JavaScript.

JS en Elementos XHTML

- Consiste en incluir código JavaScript en el código XHTML:

```
1 <html xmlns="http://www.w3.org/1999/xhtml">
2 <head>
3 <title>Ejemplo de codigo JavaScript en el propio documento</title>
4 </head>
5 <body>
6 <p onclick="alert('Un mensaje de prueba')">Presionar.</p>
7 </body>
8 </html>
```

- Los mayores inconvenientes de este método son que:
  1. 'Ensucia' el código XHTML.
  2. Complica el mantenimiento (no fomenta la reutilización).
- En general, este método sólo se utiliza para definir algunos eventos y en algunos otros casos especiales (se verá más adelante).

Primer Script

- Este primer script de ejemplo se incluye como un bloque de código dentro de una página XHTML:

```
1  <html xmlns="http://www.w3.org/1999/xhtml">
2    <head>
3    <meta http-equiv="Content-Type" content="text/html;
         charset=UTF-8" />
4    <title>El primer script</title>
5    <script type="text/javascript">
6      alert("Hola Mundo!");
7    </script>
8    </head>
9    <body>
10   <p>Esta pagina contiene el primer script</p>
11   </body>
12 </html>
```

- El script es tan sencillo que solamente incluye una sentencia:

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

- La instrucción **alert()** permite mostrar un mensaje en la pantalla del usuario.

Aprendiendo JS

- Para poder desarrollar programas y utilidades con JavaScript, es necesario conocer sus elementos básicos.
- En las siguientes secciones se explican en detalle y comenzando desde cero los conocimientos básicos necesarios para comprender y utilizar Javascript.
- Entre ellos, conceptos básicos como variables, operadores, estructuras de control y funciones.
- Y aspectos más avanzados como objetos, herencia, arrays o expresiones regulares.

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Variables I

- Una variable es un elemento que se emplea para almacenar y hacer referencia a otro valor.
- Las variables en JavaScript se crean mediante la palabra reservada **var**, ejemplo:

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

- La palabra reservada **var** solamente se debe indicar al definir por primera vez la variable o ”declarar una variable”.
- Después, cuando se utilizan las variables solamente es necesario indicar el nombre.
- Por ejemplo, lo siguiente sería un error:

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

Variables II

- 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
- Por tanto, el ejemplo anterior se puede rehacer de la siguiente manera:

```
1    var numero_1;
2    var numero_2;
3    numero_1 = 3;
4    numero_2 = 1;
5    var resultado = numero_1 + numero_2;
```

- En realialidad, en JavaSript **tampoco es necesario declarar las variables**.
- Se pueden utilizar variables que no se han definido anteriormente.

Variables III

- El ejemplo anterior también es correcto así:

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

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

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

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

Nombres de Variables

- El nombre de una variable también se conoce como **identificador** y debe cumplir las siguientes normas:
  1. Sólo puede estar formado por letras, números y los símbolos **$**(Dólar) y _(guión bajo).
  2. El primer carácter no puede ser un número.
- Las siguientes variables tienen nombres correctos:

```
1 var $numero1;
2 var _$letra;
3 var $$$otroNumero;
4 var $_a__$4;
```

- Las siguientes variables tienen identificadores incorrectos:

```
1 var 1numero;        // Empieza por un numero
2 var numero;1_123;   // Contiene un caracter ";"
```

Tipos de Variables

JavaScript divide los distintos tipos de variables en dos grupos: **tipos primitivos y objetos (tipos de referencia).**

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

Variables undefined:

- El tipo undefined corresponde a las variables que han sido definidas pero a las que aún no se les ha asignado un valor:

```
1  var variable1;
2  typeof variable1;  // devuelve undefined
```

Variables null:

- null se puede asignar a una variable como una representación de *ningún valor.*
- La variable asignada a null tiene un valor definido (null):

```
1  var nombreUsuario = null;
```

Variables Boolean

- Un Boolean sólo puede almacenar el valor true ("verdadero") o el valor false ("falso").

```
1 var variable1 = true;
2 var variable2 = false;
```

- Los valores **true** y **false** son valores especiales (no son palabras ni números ni ningún otro tipo de valor).
- Cuando es necesario convertir una variable numérica a una variable de tipo boolean, JavaScript aplica la siguiente conversión:
  - El número **0** se convierte en **false** y cualquier otro número **distinto de 0** se convierte en **true**.
- Por este motivo, en ocasiones se asocia el número **0** con el valor **false** y el número **1** con el valor **true.**

Variables Numéricas I

- En JavaScript únicamente existe un tipo de número.
- Internamente, es representado como un dato de 64 bits en coma flotante, al igual el tipo de dato double en Java.
- A diferencia de otros lenguajes de programación, no existe una diferencia entre un número entero y otro decimal, por lo que 1 y 1.0 son el mismo valor.
- Esto es significativo ya que evitamos los problemas desbordamiento en tipos de dato pequeños, al no existir la necesidad de conocer el tipo de dato.
- Si el número es entero, se indica su valor directamente.

```
1 | var variable1 = 10;
```

Variables Numéricas II

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

```
1 | var variable2 = 3.14159265;
```

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

```
1 | var variable1 = 10;
2 | var variable_octal = 034;
3 | var variable_hexadecimal = 0xA3;
```

- JavaScript define tres valores especiales muy útiles cuando se trabaja con números.
- Los valores **Infinity y –Infinity** se utilizan para representar números demasiado grandes (positivos y negativos) para JavaScript.

Variables Numéricas III

- Ejemplo de operaciones con infinitos:

```
1  var variable1 = 3, variable2 = 0;
2  console.log(variable1/variable2);  // muestra Infinity
```

- El otro valor especial definido por JavaScript es **NaN**, que es el acrónimo de "Not a Number".
- De esta forma, si se realizan operaciones matemáticas con variables no numéricas, el resultado será de tipo NaN.
- Para manejar los valores NaN, se utiliza la función relacionada **isNaN()**, que devuelve true si el parámetro que se le pasa no es un número:

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

Strings I

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

```
1 var variable1 = "hola";
2 var variable2 = 'mundo';
3 var variable3 = "hola mundo, esta es una frase mas larga";
```

- Las cadenas de texto pueden almacenar cualquier carácter, aunque algunos no se pueden incluir directamente en la declaración de la variable.

Strings II

- Si por ejemplo se incluye un ENTER para mostrar el resto de caracteres en la línea siguiente, se produce un error en la aplicación:

```
1 var variable = "hola mundo, esta es
2                                   una frase mas larga";
```

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

```
1   var variable1 = "hola 'mundo'";
2   var variable2 = 'hola "mundo"';
3   var variable3 = "hola 'mundo', esta es una "frase" mas larga";
```

Strings III

- Si el contenido de texto tiene en su interior alguna comilla simple, se encierran los contenidos con comillas dobles (como en el caso de la variable1 anterior).
- Si el contenido de texto tiene en su interior alguna comilla doble, se encierran sus contenidos con comillas simples (como en el caso de la variable2 anterior).
- Sin embargo, en el caso de la variable3 su contenido tiene tanto comillas simples como comillas dobles, por lo que su declaración provocará un error.
- Para resolver estos problemas, JavaScript define un mecanismo para incluir de forma sencilla caracteres especiales (ENTER, Tabulador) y problemáticos (comillas).

Strings IV

- Esta estrategia se denomina "mecanismo de escape", ya que se sustituyen los caracteres problemáticos por otros caracteres seguros que siempre empiezan con la barra \:
- A continuación se muestra como sería la conversión que se debe utilizar:
  - Una nueva línea: **n**
  - Un tabulador: **t**
  - Una comilla simple: \'
  - Una comilla doble: **"**
  - Una barra inclinada: \\
- Utilizando el mecanismo de escape, se pueden corregir los ejemplos anteriores:

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

Conversión entre Tipos I

- JavaScript es un lenguaje de programación “no tipado“:
  - Una misma variable puede guardar diferentes tipos de datos a lo largo de la ejecución de la aplicación.
  - De esta forma, una variable se podría inicializar con un valor numérico, después podría almacenar una cadena de texto y podría acabar la ejecución del programa en forma de variable booleana.
- No obstante, en ocasiones es necesario que una variable almacene un dato de un determinado tipo.
- Para asegurar que así sea, se puede convertir una variable de un tipo a otro, lo que se denomina **typecasting**.

Conversión entre Tipos II

- Así, JavaScript incluye un método llamado **toString()** que permite convertir variables de cualquier tipo a variables tipo cadena de texto.
- Ejemplo:

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

- JavaScript también incluye métodos para convertir los valores de las variables en valores numéricos.
- Los métodos definidos son **parseInt() y parseFloat()**, que convierten la variable que se le indica en un número entero o un número decimal respectivamente.
- La conversión numérica de una cadena se realiza carácter a carácter empezando por el de la primera posición.
- Si el primer carácter no es un número, la función devuelve el valor NaN (Not a Number).

Conversión entre Tipos III

- Si el primer carácter es un número, se continúa con los siguientes caracteres mientras estos sean números.
- Ejemplos:

```
1  var variable2 = "34";
2  parseInt(variable2);  // devuelve 34
3  var variable1 = "hola";
4  parseInt(variable1);  // devuelve NaN
5  var variable3 = "34hola23";
6  parseInt(variable3);  // devuelve 34
7  var variable4 = "34.23";
8  parseInt(variable4);  // devuelve 34
```

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

```
1  var variable1 = "hola";
2  parseFloat(variable1);  // devuelve NaN
3  var variable2 = "34";
4  parseFloat(variable2);  // devuelve 34.0
5  var variable3 = "34hola23";
6  parseFloat(variable3);  // devuelve 34.0
7  var variable4 = "34.23";
8  parseFloat(variable4);  // devuelve 34.23
```

Objetos (Tipos de Referencia)

- Los objetos en JavaScript se crean mediante la palabra reservada **new** y el nombre del prototipo que se va a instanciar.
- JavaScript no define el concepto de clase como en otros lenguajes OO (Orientados a Objetos).
- En lugar de esto, JavaScript utiliza el concepto de ”prototipo“ (que se verá más adelante).
- Por ejemplo, para crear un objeto de tipo **String** se indica lo siguiente:

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

- Los paréntesis solamente son obligatorios cuando se utilizan argumentos, aunque se recomienda incluirlos incluso cuando no se utilicen.
- JavaScript también define un prototipo para cada uno de los tipos de datos primitivos.

Objetos Boolean, Number y String I

- Existen objetos de **tipo Boolean** para las variables booleanas, **Number** para las variables numéricas y **String** para las variables de cadenas de texto.
- Estos objetos almacenan los mismos valores de los tipos de datos primitivos pero añaden propiedades y métodos.
- Ejemplo

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

- Nota sobre el prototipo String:
  - La propiedad **length** sólo está disponible en el prototipo String.
  - Por este movito, en principio no debería poder utilizarse en un dato primitivo de tipo cadena de texto.
  - Sin embargo, JavaScript convierte el tipo de dato primitivo al tipo de referencia String, obtiene el valor de la propiedad length y devuelve el resultado.
  - Este proceso se realiza de forma automática y transparente para el programador.

Objetos Boolean, Number y String II

- Nota sobre el prototipo String (cont):
  - En realidad, con una variable de tipo String no se pueden hacer muchas más cosas que con su correspondiente tipo de dato primitivo.
  - Por este motivo, no existen muchas diferencias prácticas entre utilizar el tipo de referencia o el tipo primitivo.
  - Salvo en el caso del resultado del operador **typeof** y en el caso de la función **eval()**, como se verá más adelante.
- Valor vs. referencia:
  - La principal diferencia entre los tipos de datos es que los datos primitivos se manipulan por valor y los tipos de referencia se manipulan por referencia.
  - Los conceptos "por valor" y "por referencia" son iguales al resto de lenguajes de programación (ejemplo C).
  - Aunque Javascript tiene particularidades como por ejemplo que no existe el concepto de puntero.

Variables por Valor I

- Cuando un dato se manipula por valor, lo único que importa es el valor en sí.
- Cuando se asigna una variable por valor a otra variable, se copia directamente el valor de la primera variable en la segunda.
- Cualquier modificación que se realice en la segunda variable es independiente de la primera variable.
- De la misma forma, cuando se pasa una variable por valor a una función (como se explicará más adelante) sólo se pasa una copia del valor.
- Así, cualquier modificación que realice la función sobre el valor pasado no se refleja en el valor de la variable original.

Variables por Valor II

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

```
1 var variable1 = 3;
2 var variable2 = variable1;
3 variable2 = variable2 + 5;
4 // Ahora variable2 = 8 y variable1 sigue valiendo 3
```

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

Variables por Referencia


```
1 // variable1 = 25 diciembre de 2009
2 var variable1 = new Date(2009, 11, 25);
3 // variable2 = 25 diciembre de 2009
4 var variable2 = variable1;
5
6 // variable2 = 31 diciembre de 2010
7 variable2.setFullYear(2010, 11, 31);
8 // Ahora variable1 tambien es 31 diciembre de 2010
```

En el ejemplo anterior, se utiliza un tipo de dato de referencia denominado Date y que se utiliza para manejar fechas:

- Se crea una variable llamada variable1 y se inicializa la fecha a 25 de diciembre de 2009.
- Al constructor del objeto Date se le pasa el año, el número del mes (siendo 0 = enero, 1 = febrero, ..., 11 = diciembre) y el día (al contrario que el mes, los días no empiezan en 0 sino en 1).
- A continuación, se asigna el valor de la variable1 a otra variable llamada variable2.
- Como Date es un tipo de referencia, la asignación se realiza por referencia.
- Por lo tanto, las dos variables quedan "unidas" y hacen referencia al mismo objeto, al mismo dato de tipo Date.
- De esta forma, si se modifica el valor de variable2 (y se cambia su fecha a 31 de diciembre de 2010) el valor de variable1 se verá automáticamente modificado.

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Operadores

- Hasta ahora, sólo se ha visto cómo crear variables de diferentes tipos y cómo mostrar su valor mediante la función **alert()**.
- Para hacer programas realmente útiles, son necesarias otro tipo de herramientas.
- 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.

Operador Asignación

- El operador de asignación se utiliza para guardar un valor específico en una variable.
- El símbolo utilizado es =:

```
1  var numero1 = 3;
```

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

```
1  var numero1 = 3;
2  var numero2 = 4;
3
4  /* Error, la asignacion siempre se realiza a una variable,
5  por lo que en la izquierda no se puede indicar un numero */
6  5 = numero1;
7
8  // Ahora, la variable numero1 vale 5
9  numero1 = 5;
10
11 // Ahora, la variable numero1 vale 4
12 numero1 = numero2;
```

Operadores Aritméticos I

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

```
1  var numero1 = 10;
2  var numero2 = 5;
3  resultado = numero1 / numero2;          // resultado = 2
4  resultado = numero1 + numero2;          // resultado = 15
5  resultado = numero1 - numero2;          // resultado = 5
6  resultado = numero1 * numero2;          // resultado = 50
7  resultado = numero1 % numero2;          // resultado = 0
8  numero1 = 9;
9  numero2 = 5;
10 resultado = numero1 % numero2;           // resultado = 4
```

- También se pueden se pueden combinar con el operador asignación:

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

Operadores Aritméticos II

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

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

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

```
1 var numero = 5;
2 numero = numero + 1;
3 alert(numero);  // numero = 6
```

- De forma equivalente, el operador decremento (indicado como un prefijo **- -** en el nombre de la variable).

Operadores Aritméticos III

- Se utiliza para decrementar el valor de la variable:

```
1 var numero = 5;
2 --numero;
3 alert(numero);  // numero = 4
```

- El anterior ejemplo es equivalente a:

```
1 var numero = 5;
2 numero = numero - 1;
3 console.log(numero);  // numero = 4
```

- Los operadores de incremento y decremento no solamente se pueden indicar como prefijo del nombre de la variable, sino que también es posible utilizarlos como sufijo:

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

- El resultado de ejecutar el script anterior es el mismo que cuando se utiliza el operador ++numero.

Operadores Aritméticos IV

- El siguiente ejemplo muestra las diferencias entre un pre-incremento o un post-incremento:

```
1  //ejemplo 1
2  var numero1 = 5;
3  var numero2 = 2;
4  numero3 = numero1++ + numero2;
5  // numero3 = 7, numero1 = 6
6  //ejemplo 2
7  var numero1 = 5;
8  var numero2 = 2;
9  numero3 = ++numero1 + numero2;
10 // numero3 = 8, numero1 = 6
```

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

Operadores Lógicos I

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

Operadores Lógicos II

1 *Negación:*

- La negación lógica se obtiene prefijando el símbolo **!** al identificador de la variable.
- Se utiliza para obtener el valor contrario al valor de la variable:

```
1  var visible = true;
2  alert(!visible);  // Muestra "false"
```

- 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 otro caso.
- Ejemplos:

```
1  var cantidad = 0;
2  vacio = !cantidad;  // vacio = true
3
4  cantidad = 2;
5  vacio = !cantidad;  // vacio = false
6
7  var mensaje = "";
8  mensajeVacio = !mensaje;  // mensajeVacio = true
9
10 mensaje = "Bienvenido";
11 mensajeVacio = !mensaje;  // mensajeVacio = false
```

Operadores Lógicos III

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

```
1  var valor1 = true;
2  var valor2 = false;
3  resultado = valor1 && valor2; // resultado = false
4
5  valor1 = true;
6  valor2 = true;
7  resultado = valor1 && valor2; // resultado = true
```

Operadores Lógicos IV

3. ***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:

```
1 var valor1 = true;
2 var valor2 = false;
3 resultado = valor1 || valor2; // resultado = true
4
5 valor1 = false;
6 valor2 = false;
7 resultado = valor1 || valor2; // resultado = false
```

Operadores Relacionales I

- 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 (!=).
- El resultado de todos estos operadores siempre es un valor booleano. Ejemplos:

```
1  var numero1 = 3;
2  var numero2 = 5;
3  resultado = numero1 > numero2; // resultado = false
4  resultado = numero1 < numero2; // resultado = true
5  numero1 = 5;
6  numero2 = 5;
7  resultado = numero1 >= numero2; // resultado = true
8  resultado = numero1 <= numero2; // resultado = true
9  resultado = numero1 == numero2; // resultado = true
10 resultado = numero1 != numero2; // resultado = false
```

Operadores Relacionales II

- Se debe tener especial cuidado con el operador de igualdad (==).
- El operador **==** se utiliza para comparar el valor de dos variables, por lo que es muy diferente del operador **=**, que se utiliza para asignar un valor a una variable:

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

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

Operador typeof I

- El operador **typeof** se emplea para determinar el tipo de dato que almacena una variable.
- Su uso es muy sencillo, ya que sólo es necesario indicar el nombre de la variable cuyo tipo se quiere averiguar:

```
1  var myFunction = function() {
2    console.log('hola');
3  };
4
5  var myObject = {
6      foo : 'bar'
7  };
8
9  var myArray = [ 'a', 'b', 'c' ];
10
11 var myString = 'hola';
12
13 var myNumber = 3;
14
15 typeof myFunction;   // devuelve 'function'
16 typeof myObject;     // devuelve 'object'
17 typeof myArray;      // devuelve 'object' -- tenga cuidado
18 typeof myString;     // devuelve 'string'
19 typeof myNumber;     // devuelve 'number'
```

Operador typeof II


```
20
21  typeof null;          // devuelve 'object' -- tenga cuidado
22
23  if (myArray.push && myArray.slice && myArray.join) {
24      // probablemente sea un vector
25      // (este estilo es llamado, en ingles, "duck typing")
26  }
27
28  if (Object.prototype.toString.call(myArray) === '[object Array]')
        {
29      // definitivamente es un vector;
30      // esta es considerada la forma mas robusta
31      // de determinar si un valor es un vector.
32  }
```

- Los posibles valores de retorno del operador son: undefined, boolean, number, string para cada uno de los tipos primitivos y object para los valores de referencia y también para los valores de tipo null.
- El operador typeof no distingue entre las variables declaradas pero no inicializadas y las variables que ni siquiera han sido declaradas.

Operador instaceof

- El operador typeof no es suficiente para trabajar con tipos de referencia, ya que devuelve el valor object para cualquier objeto independientemente de su tipo.
- Por este motivo, JavaScript define el operador instanceof para determinar el prototipo concreto de un objeto.

```
1  var variable1 = new String("hola mundo");
2  typeof variable1;              // devuelve "object"
3  variable1 instanceof String;   // devuelve true
```

- El operador **instanceof** sólo devuelve como valor true o false.
- De esta forma, instanceof no devuelve directamente el prototipo con el que se ha instanciado la variable, sino que se debe comprobar cada posible prototipo individualmente.

Outline

JavaScript

1. JavaScript
   - Introducción
   - Variables
   - Operadores
   - Estructuras de control
   - Funciones
   - Objetos
   - Herencia
   - Arrays
   - Utilidades
   - JSON
   - DOM
   - Eventos

Estructuras de control

- Los programas que se pueden realizar utilizando solamente variables y operadores son una simple sucesión lineal de instrucciones básicas.
- Las estructuras de control de flujo son instrucciones del tipo "si se cumple esta condición, hazlo; si no se cumple, haz esto otro".
- También existen instrucciones del tipo "repite esto mientras se cumpla esta condición".
- De esta manera los programas dejan de ser una sucesión lineal de instrucciones para convertirse en programas inteligentes que pueden tomar decisiones en función del valor de las variables.

Estructura if... else I

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

```
1 if(condicion) {
2 ...
3 }
```

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

Estructura if... else II

- Ejemplo:

```
1 var showMsg = true;
2 if(showMsg) {
3 alert("Hola Mundo");
4 }
```

- En el ejemplo anterior, el mensaje sí que se muestra al usuario ya que la variable showMsg tiene un valor de true y por tanto, el programa entra dentro del bloque de instrucciones del if.
- El ejemplo se podría reescribir también como:

```
1 var showMsg = true;
2
3 if(showMsg == true) {
4 console.log("Hola Mundo");
5 }
```

- En este caso, la condición es una comparación entre el valor de la variable showMsg y el valor true.

Estructura if... else III

- Confundir los operadores == y = suele ser el origen de muchos errores de programación: las comparaciones siempre se realizan con == ya que el operador = asigna valores. Ejemplo:

```
1  var mostrarMensaje = true;
2
3  // Se comparan los dos valores
4  if(mostrarMensaje == false) {
5  ...
6  }
7
8  // Error - Se asigna el valor "false" a la variable
9  if(mostrarMensaje = false) {
10 ...
11 }
```

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

```
1  var mostrado = false;
2  if(!mostrado) {
3  alert("Es la primera vez que se muestra el mensaje");
4  }
```

Estructura if... else IV

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

```
1 var mostrado = false;
2 var usuarioPermiteMensajes = true;
3
4 if(!mostrado && usuarioPermiteMensajes) {
5 alert("Es la primera vez que se muestra el mensaje");
6 }
```

- La condición anterior está formada por una operación AND sobre dos variables.
- A su vez, a la primera variable se le aplica el operador de negación antes de realizar la operación AND.
- De esta forma, como el valor de mostrado es false, el valor *!mostrado* sería true.
- Como la variable *usuarioPermiteMensajes* vale true, el resultado de *!mostrado* && *usuarioPermiteMensajes* sería igual a true && true.

Estructura if... else V

- Por lo que el resultado final de la condición del *if()* sería true y por tanto, se ejecutan las instrucciones que se encuentran dentro del bloque del *if()*.
- En ocasiones, las decisiones que se deben realizar no son del tipo "si se cumple la condición, hazlo; si no se cumple, no hagas nada".
- Normalmente las condiciones suelen ser del tipo "si se cumple esta condición, hazlo; si no se cumple, haz esto otro".
- Para este segundo tipo de decisiones, existe una variante de la estructura if llamada **if...else**. Su definición formal es la siguiente:

```
1 if(condicion) {
2 ...
3 }
4 else {
5 ...
6 }
```

Estructura if... else VI

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

```
1 var edad = 18;
2 if(edad >= 18) {
3 alert("Eres mayor de edad");
4 }
5 else {
6 alert("Eres menor de edad");
7 }
```

- Así mismo, podemos comparar variables de tipo cadenas de texto:

```
1 var nombre = "";
2 if(nombre == "") {
3 alert("Aun no nos has dicho tu nombre");
4 }
5 else {
6 alert("Hemos guardado tu nombre");
7 }
```

Estructura if... else VII

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

```
1  var edad = 18;
2  if(edad < 12) {
3  alert("Eres muy joven");
4  }
5  else if(edad < 19) {
6  alert("Eres un adolescente");
7  }
8  else if(edad < 35) {
9  alert("Sigues siendo joven");
10 }
11 else {
12 alert("Eres un adulto.");
13 }
```

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

Estructura switch I

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

```
1  if(dia == 1) {
2    console.log("Hoy es lunes.");
3  } else if(dia == 2) {
4    console.log("Hoy es martes.");
5  } else if(dia == 3) {
6    console.log("Hoy es miercoles.");
7  } else if(dia == 4) {
8    console.log("Hoy es jueves.");
9  } else if(dia == 5) {
10   console.log("Hoy es viernes.");
11 } else if(dia == 6) {
12   console.log("Hoy es sabado.");
13 } else if(dia == 0) {
14   console.log("Hoy es domingo.");
15 }
```

- En este caso es más conveniente utilizar una estructura de control de tipo switch, ya que permite ahorrarnos trabajo y producir un código más limpio.

Estructura switch II

- Un ejemplo de uso es el siguiente:

```
1  switch(dia) {
2  case 1: console.log("Hoy es lunes."); break;
3  case 2: console.log("Hoy es martes."); break;
4  case 3: console.log("Hoy es miercoles."); break;
5  case 4: console.log("Hoy es jueves."); break;
6  case 5: console.log("Hoy es viernes."); break;
7  case 6: console.log("Hoy es sabado."); break;
8  case 0: console.log("Hoy es domingo."); break;
9  default: console.log("something"); break;
10 }
```

- 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: {...}
- 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.

Estructura switch III

- 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.
- 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 I

- La estructura **while** permite crear bucles que se ejecutan cero o más veces, dependiendo de la condición indicada.
- Su definición formal es:

```
1 while(condicion) {
2 ...
3 }
```

- La idea del funcionamiento del bucle while es la siguiente:
  - "Mientras la condición indicada se siga cumpliendo, repite la ejecución de las instrucciones definidas dentro del while".
  - Si la condición no se cumple ni siquiera la primera vez, el bucle no se ejecuta.
  - Si la condición se cumple, se ejecutan las instrucciones una vez y se vuelve a comprobar la condición.
  - Si se sigue cumpliendo la condición, se vuelve a ejecutar el bucle y así se continúa hasta que la condición no se cumpla.
- Evidentemente, las variables que controlan la condición deben modificarse dentro del propio bucle, ya que de otra forma, la condición se cumpliría siempre y el bucle while se repetiría indefinidamente.

Estructura while II

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

```
1  var resultado = 0;
2  var numero = 100;
3  var i = 0;
4
5  while(i <= numero) {
6    resultado += i;
7    i++;
8  }
9
10 alert(resultado);
```

  - Es decir, mientras que la variable i sea menor o igual que la variable numero, se ejecutan las instrucciones del bucle.
  - Dentro del bucle se suma el valor de la variable i al resultado total (variable resultado) y se actualiza el valor de la variable i, que es la que controla la condición del bucle.
  - Si no se actualiza el valor de la variable i, la ejecución del bucle continua infinitamente o hasta que el navegador permita al usuario detener el script.

Estructura for I

- La estructura **for** permite realizar repeticiones de un bloque de código de forma sencilla (también llamado bucle).
- Su definición formal es la siguiente:

```
1 for(inicializacion; condicion; actualizacion) {
2 ...
3 }
```

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

Estructura for II

- 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.

- Ejemplo:

```
1  var dias = ["Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "
       Sabado", "Domingo"];
2
3  for(var i=0; i<7; i++) {
4  alert(dias[i]);
5  }
```

- El ejemplo muestra los días de la semana contenidos en un vector o array.
- Nota. El uso del array en el ejemplo es auto-explicativo (posteriormente se proporcionan más detalles sobre arrays).

Estructura for...in I

- Una estructura de control derivada de for es **for...in**.
- Esta estructura permite interaccionar con objetos Javascript iterativamente.
- En particular, for...in permite recorrer las propiedades de un objeto.
- En cada iteración, un nuevo nombre de propiedad del objeto es asignada a una variable de control del bucle.
- Ejemplo:

```
1 for(propiedad in object) {
2 if (object.hasOwnProperty(propiedad)) {
3   ...
4 }
5 }
```

- Suele ser conveniente comprobar que la propiedad pertenece efectivamente al objeto, a través de la función **object.hasOwnProperty(propiedad)**.

Estructura for...in II

- Javascript también adapta esta estructura para los arrays.
- La definición de la estructura adaptada a arrays es la siguiente:

```
1 for(indice in array) {
2 ...
3 }
```

- Si se quieren recorrer todos los elementos que forman un array, la estructura for...in es la forma más eficiente de hacerlo.
- Ejemplo:

```
1 var dias = ["Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "
      Sabado", "Domingo"];
2
3 for(i in dias) {
4 alert(dias[i]);
5 }
```

Estructura for...in III

- La variable que se indica como indice es la que se puede utilizar dentro del bucle for...in para acceder a los elementos del array.
- De esta forma, en la primera repetición del bucle la variable i vale 0 y en la última vale 6.
- Esta estructura de control es la más adecuada para recorrer arrays y objetos:
  - Evita tener que indicar la inicialización y las condiciones del bucle.
  - Funciona correctamente cualquiera que sea la longitud del array/objeto.
  - De hecho, sigue funcionando igual aunque varíe el número de elementos del array.

Estructura try...catch

- La estructura **try... catch** controla las excepciones que se puedan producir en la ejecución de un bloque de código.
- Las excepciones son situaciones de error.
- La definición formal de la estructura es la siguiente:

```
1 try {
2 funcion_que_no_existe();
3 } catch(ex) {
4 console.log("Error detectado: " + ex.description);
5 }
```

- En este ejemplo, llamamos a una función que no está definida y esto provoca una excepción.
- Este error o excepción es capturado por la cláusula **catch**.
- catch contiene una serie de sentencias que indican las acciones a realizar con esa excepción que acaba de producirse.
- Si no se produce ninguna excepción en el bloque try, no se ejecuta el bloque dentro de catch.

Cláusula finally

- La cláusula **finally** contiene las sentencias a ejecutar después de los bloques try...catch.
- Las sentencias incluidas en este bloque se ejecutan siempre, se haya producido una excepción o no.
- Un uso clásico de la cláusula finally es el de liberar recursos que el script ha solicitado.
- Ejemplo:

```
1  abrirFichero()
2  try {
3    escribirFichero(datos);
4  } catch(ex) {
5  // Tratar la excepcion
6  } finally {
7    cerrarFichero(); // siempre se cierra el recurso
8  }
```

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Funciones I

- Las funciones son la base de la modularidad de cualquier lenguaje de programación.
- Son utilizadas para reutilizar código, ocultar información y para la abstracción.
- **Llamar a una función** suspende la ejecución de la función actual, pasando el control y los parámetros a la nueva función.
- Las funciones en JavaScript se definen mediante la palabra reservada **function**, seguida del nombre de la función.
- El nombre de la función se utiliza para llamar a esa función cuando sea necesario (igual que con las variables, a las que se les asigna un nombre único).
- Después del nombre de la función, se incluyen dos paréntesis donde indicaremos los parámetros de la función.
- Por último, los símbolos { y } se utilizan para encerrar todas las instrucciones que pertenecen a la función.

Funciones II

- La definición formal es la siguiente:

```
1 function nombre_funcion() {
2 ...
3 }
```

- Las funciones en Javascript **son objetos**.
- Pueden ser utilizadas como cualquier otro objeto.
- Pueden ser almacenadas en variables, objetos o arrays.
- Además, al ser objetos, también pueden tener métodos.
- Lo que las hace realmente especiales es que **pueden ser llamadas**.
- Pueden ser pasadas como argumentos a otras funciones, y pueden ser retornadas por otras funciones.

Argumentos I

- Las funciones más sencillas no necesitan ninguna información para producir sus resultados.
- Sin embargo, es muy habitual que las funciones utilicen información externa en forma de **argumentos**.
- Antes de poder utilizar los argumentos, la función debe indicar cuántos argumentos necesita y cuál es el nombre de cada argumento.
- Entonces, cuando se llama a la función, se incluyen los valores (o expresiones) de los argumentos.
- Cuando se define la función, los argumentos se indican dentro de los paréntesis que van detrás del nombre de la función.
- Los diferentes argumentos van separados por comas:

```
1 var s = function sumShow(n1, n2) { ... }
```

Argumentos II

- Entonces, para utilizar el valor de los argumentos dentro de la función, se debe emplear el mismo nombre con el que se definieron.
- Ejemplo:

```
1 var s = function sumShow(n1, n2) {
2 var resultado = n1 + n2;
3 console.log("El resultado es " + resultado);
4 }
```

- Dentro de la función, el valor de la variable n1 será igual al primer valor que se le pase a la función y el valor de la variable n2 será igual al segundo valor que se le pasa.
- Las funciones no solamente puede recibir variables y datos, sino que también pueden devolver los valores que han calculado.

Retorno I

- Para devolver valores dentro de una función se utiliza la palabra reservada **return**.
- Las funciones pueden devolver **un solo valor cada vez que se ejecutan**.
- Aunque, el valor devuelto puede ser de cualquier tipo.
- Cuando la función llega a una instrucción de tipo **return**, se devuelve el valor indicado y finaliza la ejecución de la función.
- Por tanto, si hay instrucciones después de un return se ignoran (por eso return suele ser la última instrucción en la mayoría de funciones).
- Para recoger el valor de vuelto por una función debe indicarse el nombre de una variable en el punto en el que se realiza la llamada.

Retorno II

- Ejemplo:

```
1 var calcular_precio = function (precio) {
2 var impuestos = 1.21;
3 var gastosEnvio = 10;
4 var precioTotal = ( precio * impuestos ) + gastosEnvio;
5
6 return precioTotal;
7 }
```

- En el ejemplo anterior, la ejecución de la función llega a la instrucción `return precioTotal;`
- En ese momento se devuelve el valor que contenga la variable `precioTotal`.
- Para recoger el valor devuelto por la función en una variable denominada `precio_calculado` debemos utilizar la siguiente sentencia:

```
1 var precio_calculado = calcular_precio(23.34);
```

- Si no se indicara ninguna ninguna variable para recoger el resultado JavaScript no muestra ningún error (el valor devuelto por la función simplemente se pierde).

Arguments I

- Además de los argumentos declarados, todas las funciones en JavaScript reciben dos argumentos extra:
  1. `this`: sirve para referirse al objeto de la propia función (lo veremos más adelante).
  2. `arguments`: es un pseudo-array cuyo funcionamiento se detalla a continuación.
- `arguments` da a la función acceso a todas los argumentos pasados en la llamada:
  - Esto incluye argumentos extra que no coinciden con los parámetros definidos en la función.
  - Esto además hace posible escribir funciones que toman un número indefinido de parámetros.

Arguments II

- Ejemplo de uso de `arguments`:

```
1  var sum = function() {
2      var i, sum = 0;
3      for (i = 0; i < arguments.length; i += 1) {
4        sum += arguments[i];
5      }
6      return sum;
7  };
8  console.log("El resultado de la suma es" + sum(4, 8, 15, 16, 23, 42)
       ); // 108
```

- Nota:
  - Debido a un problema de diseño, `arguments` no es realmente un array.
  - En realidad es un objeto que se comporta ”como un array“.
  - Dispone de la propiedad length, pero no incluye el resto de métodos de los arrays.

Funciones en Variables y Argumentos

- En JavaScript, las funciones son objetos y como tales:
  - Pueden ser asignadas a variables.
  - Pueden ser pasadas a otras funciones como argumentos.
- Ejemplo:

```
1  // Funcion que utiliza como parametro otra funcion:
2  var myFn = function(fn) {
3    var result = fn();
4    console.log(result);
5  };
6
7  // **Primera forma de llamar a myFn:
8  myFn(function() { return 'hola mundo'; });  // muestra 'hola mundo'
9
10 // **Segunda forma de llamar a myFn:
11 var myOtherFn = function() {
12     return 'hola mundo';
13 };
14 myFn(myOtherFn); // muestra 'hola mundo'
```

Alcance I

- El alcance en un lenguaje de programación controla la visibilidad y el ciclo de vida de las variables y los parámetros.
- La mayoría de los lenguajes, con sintaxis de C, tienen un alcance de bloque (sentencias definidas entre llaves).
- Es decir, todas las variables definidas en un bloque no son visibles fuera de ese bloque.
- Desafortunadamente, JavaScript no tiene esa visibilidad de bloque, a pesar de que su sintaxis así pueda sugerirlo.
- JavaScript tiene un alcance de función:
  - Los parámetros y variables definidos dentro de una función no son visibles fuera de esa función.
  - Una variable definida en cualquier lugar de la función, es visible desde cualquier lugar dentro de esa función.

Alcance II

- Ejemplo:

```
1  var foo = function () {
2    var a = 3, b = 5;
3    var bar = function () {
4        var b = 7, c = 11;
5        // En este punto, a es 3, b es 7, y c es 11
6
7        a += b + c;
8        // En este punto, a es 21, b es 7, y c es 11
9    };
10
11   // En este punto, a es 3, b es 5, y c es undefined
12
13   bar();
14
15   // En este punto, a es 21, b es 5
16 };
```

- Nota. Si no se tiene en cuenta que el alcance en JavaScript es a nivel de nivel de función y no de bloque, esto puede ser una fuente de problemas.

Callbacks

- Las funciones pueden facilitar el trabajo con métodos asíncronos.
- Supongamos el siguiente caso en el que hacemos una petición al servidor:

```
1 request = prepare_the_request();
2 response = send_request_synchronously(request);
3 display(response);
```

- El problema aquí, es que en la línea 2 esperamos la respuesta del servidor, por lo que bloqueamos la ejecución hasta obtener una respuesta.
- Una estrategia mucho mejor es realizar una llamada asíncrona, proporcionando una función de respuesta (callback) que se ejecutará cuando la respuesta esté disponible:

```
1 request = prepare_the_request();
2 send_request_asynchronously(request, function (response) {
3     display(response);
4 });
```

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Objetos

- Excepto los tipos primitivos (*undefined, null, boolean, number y string.*) el resto de elementos en JavaScript son objetos: funciones, arrays, expresiones regulares, etc.
- Un objeto en JavaScript es un contenedor de propiedades, donde una propiedad tiene un nombre y un valor.
- El nombre de una propiedad puede ser una cadena de caracteres, incluso una vacía.
- El valor de la propiedad puede ser cualquier valor que podamos utilizar en JavaScript, excepto *undefined.*
- En JavaScript, los objetos son básicamente *tablas hash*, esto es, un grupo de propiedades y funciones que pueden ser accedidos a través de una clave.
- Tanto las propiedades como los métodos (que no son más que propiedades cuyo valor es una función) pueden ser creados dinámicamente en tiempo de ejecución.

Creación de Objetos I

- La manera más simple de construir un objeto es:
  - Declarar una nueva variable de tipo **Object**.
  - Asignarle las propiedades o métodos que el objeto necesite.
- Ejemplo:

```
1 var obj = new Object();
2 obj.foo = "bar";
3 obj.hello = function() { console.log("Hello world!"); }
```

- Otra forma, denominada ”notación literal“, es crear el objeto utilizando una pareja de llaves { }:

```
1   var obj   = {};
2   obj.foo   = "bar";
3   obj.hello = function() { console.log("Hello world!"); }
```

- Se pueden definir las propiedades entre las llaves:

```
1 var obj = {
2     "foo"   : "bar",
3     "hello" : function() { console.log("Hello world!"); }
4 };
```

Creación de Objetos II

- Respecto al **nombre de una propiedad**:
  - Las comillas dobles alrededor del nombre de la propiedad no son necesarias si el nombre es un nombre legal de JavaScript (es decir, no es una palabra reservada).
  - El nombre de una propiedad es un string (incluso puede ser un string vacío).
- Respecto al **valor de una propiedad**, puede ser cualquier expresión, incluso la definición de otro objeto:

```
1  var flight = {
2    airline: "Oceanic",
3    number: 815,
4    departure: {
5        IATA: "SYD",
6        time: "2004-09-22 14:55",
7        city: "Sydney"
8    },
9    arrival: {
10       IATA: "LAX",
11       time: "2004-09-23 10:42",
12       city: "Los Angeles"
13   }
14 };
```

Métodos

- En JavaScript, los objetos son dinámicos, esto quiere decir que sus propiedades no tienen por qué definirse en el momento en el que creamos el objeto.
- Podemos añadir nuevas propiedades al objeto en tiempo de ejecución, tan solo indicando el nombre la propiedad y asignándole un valor o función.

```
1 var cat = new Object();
2 cat.name = "Rufus";
3 cat.species = "cat";
4 cat.hello = function() { console.log("miaow"); }
```

- En el ejemplo, el valor de la propiedad ”hello“ no es un tipo primitivo sino una función.
- Las propiedades que tienen como valor una función, son tratadas como métodos del objeto.
- Notar que no es necesario definir la función previamente (se puede utilizar una función anónima para declarar el método).

Usos de la Notación Literal I

- Crear objetos utilizando la notación literal es muy útil en situaciones en las que queremos pasar una serie de datos relacionados a una función o configurar una librería externa.
- Ejemplo:
  - Imaginemos una situación en la que utilizamos una librería que nos permite crear una galería de imágenes.
  - Una posible solución sería definir un método para cada una de las propiedades de la galería, de manera que pudiésemos llamar a cada uno de esos métodos con el valor correspondiente.
  - Esto supondría tener dos métodos por cada propiedad (getter y setter), así como una llamada para cada propiedad que deseemos configurar.
  - Utilizando una notación literal de objetos, podemos ahorrarnos todo este código y configurar la librería utilizando un único objeto.

Usos de la Notación Literal II

- Creación del objeto:

```
1  domElement.fancybox({
2    maxWidth    : 800,
3    maxHeight   : 600,
4    fitToView   : false,
5    width       : '70%',
6    height      : '70%',
7    autoSize    : false,
8    closeClick  : false,
9    openEffect  : 'none',
10   closeEffect : 'none'
11 });
```

- Otra de las ventajas que nos ofrece esta notación literal, es poder crear librerías de utilidades:
  - Esto es equivalente a crear una clase abstracta con métodos estáticos en otros lenguajes de programación como Java o C#.

Usos de la Notación Literal III

- A continuación se muestra un ejemplo de cómo fácilmente podríamos crear una librería de utilidades para arrays:

```
1  var ArrayUtil = {
2    contains : function(array, element)
3    {
4        for(var x=0; x<array .length; x++)
5        {
6    if(array[x] === element)
7        return true;
8        }
9        return false;
10   },
11   exclude : function(list, items)
12   {
13       ...
14   },
15   makeList : function(list)
16   {
17       ...
18   }
19 }
20
21 var list = ["A", "B", "C"];
22 console.log("Contiene A? " + ArrayUtil.contains(list, "A"));
```

Acceso a Propiedades I

- En general, para acceder al valor de una propiedad de un objeto se indica el nombre de la propiedad dentro de corchetes [ ] (como si accediésemos a un elemento de un *array*).
- Si el nombre de la propiedad es un nombre legal en JavaScript (no palabra reservada) se puede utilizar la notación con punto para acceder al valor de las propiedades.
- Ejemplos:

```
1 flight["airline"]        // "Oceanic"
2 flight.departure.IATA    // "SYD"
```

- En cualquier caso, si es posible, es preferible utilizar la notación con punto ya que es más corta y comúnmente utilizada en otros lenguajes orientados a objetos.

Acceso a Propiedades II

- En algunas ocasiones, es interesante utilizar la notación con corchetes que utiliza strings como claves:
  - Esta notación nos permite construir un string en tiempo de ejecución y utilizarlo para acceder a una determinada propiedad de un objeto.
  - Por ejemplo, podemos crear un bucle para iterar sobre las propiedades de un objeto:

```
1  var myObject = {
2      property1: "chocolate",
3      property2: "cake",
4      property3: "brownies"
5  }
6
7  for(var x=1; x<4; x++){
8    console.log(myObject["property" + x]);
9    }
10
11 // Alternativamente:
12 for(var property in myObject){
13   console.log(cat[property].toString());
14   }
```

Acceso a Propiedades III

- Si se intenta acceder a una propiedad que no existe, JavaScript devuelve el valor ”undefined“:

```
1 var a = myObject.property4     // a contiene el valor undefined.
```

- Asignar un valor si una propiedad no está definida:

```
1 var a = myObject["property4"] || "red";
```

  - El operador || asigna el valor `red` a la variable `a` si `myObject["property4"]` es un ”falsey“.
  - Un falsey es un valor false, null, undefined, 0 o "".

- Por otra parte, al acceder a una propiedad undefined:
  - Se lanzará una excepción de tipo **TypeError**.
  - En ese caso, el operador && puede usarse para asegurarnos que la propiedad existe y es accesible:

```
1 a = flight.equipment        // a es undefined
2 a = flight.equipment.model // TypeError: Cannot read
3                             // property 'model' of undefined
4 a = flight.equipment && flight.equipment.model  // undefined
```

Modificación

- El valor de una objeto puede actualizarse a través de una asignación.
- Si el nombre de la propiedad existe en el objeto, su valor es reemplazado:

```
1 | stooge['first-name'] = 'Arkaitz';
```

- Si la propiedad no existe en el objeto, esta nueva propiedad es añadida al objeto:

```
1 | stooge['middle-name'] = 'Lester';
2 | stooge.nickname = 'Curly';
3 | flight.equipment = { model: 'Boeing 777' }
4 | flight.status = 'overdue';
```

Borrado

- El operador delete puede ser utilizado para eliminar la propiedad de un objeto.
- Este operador eliminará la propiedad de un objeto, si la tuviera, pero no afectará al resto de propiedades de los prototipos.

```
1 another_stooge.nickname // 'Moe'
2
3 // Remove nickname from another_stooge, revealing
4 // the nickname of the prototype.
5 delete another_stooge.nickname;
6
7 another_stooge.nickname // 'Curly'
```

Referencias

- Los objetos siempre son accedidos como referencias.
- Nunca se copia su valor cuando los asignamos a otros objetos o los pasamos como parámetros en funciones.
- Ejemplos:

```
1  var x = y;
2  x.nickname = 'Curly';
3  var nick = y.nickname; // nick contiene 'Curly' porque x e y
4                         // referencian al mismo objeto
5
6  var a = {}, b = {}, c = {};  // a, b, y c hacen referencia a
7                               // diferentes objetos vacios
8
9  a = b = c = {};   // a, b, y c hacen referencia al
10                   // mismo objeto vacio
11
12 var obj { value = 5 };
13 console.log(obj.value); // o.value = 5
14
15 function change(obj)
16 {
17     obj.value = 6;
18 }
19 change(obj);
20 console.log(obj.value); // o.value = 6
```

Enumeración de Propiedades I

- La sentencia **for...in** puede iterar sobre todos los nombres de propiedades de un objeto.
- Esta iteración incluirá todas las propiedades, funciones y propiedades definidas en los prototipos, en las que no podemos estar interesados.
- La mejor manera de filtrar estas propiedades es a través de la función **hasOwnProperty** y el operador **typeof**:

```
1  var name;
2  for (name in myobj) {
3      if (typeof myobj[name] !== 'function') {
4  console.log(name + ': ' + myobj[name]);
5      }
6  }
```

Enumeración de Propiedades II

- El problema del código anterior es que no hay garantía del orden en el que se van a mostrar las propiedades.
- Si ese orden es importante, es mejor no utilizar for...in.
- En este caso, lo mejor es acceder directamente a las propiedades concretas en el orden que definamos:

```
1  var i;
2  var properties = [
3      'first-name',
4      'middle-name',
5      'last-name',
6      'profession'
7  ];
8
9  for (i = 0; i < properties.length; i += 1) {
10     console.log(properties[i] + ': ' +
11 myobj[properties[i]]);
12 }
```

Variables Globales I

- JavaScript permite crear variables globales de una manera muy sencilla.
- El principal problema de las variables globales es que pueden causar fácilmente conflictos de nombres.
- Sobretodo si nuestro código utilizan librerias, widgets u otras aplicaciones.
- Una manera de solucionar esto es crear una única variable global con el nombre de nuestra aplicación.

```
1 var MYAPP = {};
```

- En este caso, la variable global MYAPP va a incluir el resto de variables de nuestra aplicación.

Variables Globales II


```
1  MYAPP.stooge = {
2    "first-name": "Joe",
3    "last-name": "Howard"
4  };
5
6  MYAPP.flight = {
7      airline: "Oceanic",
8      number: 815,
9      departure: {
10 IATA: "SYD",
11 time: "2004-09-22 14:55",
12 city: "Sydney"
13     },
14     arrival: {
15 IATA: "LAX",
16 time: "2004-09-23 10:42",
17 city: "Los Angeles"
18     }
19 };
```

- Con una sola variable global, además, nuestra aplicación puede leerse y entenderse de manera más sencilla.

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Prototipos

- JavaScript en un lenguaje orientado a objetos **basado en prototipos**, en lugar de clases como son otros lenguajes OO.
- Debido a esto, probablemete es menos evidente entender cómo JavaScript nos permite crear herencia entre objetos.
- En primer lugar, hay que comentar que **todos los objetos de JavaScript enlazan con un objeto prototipo del que heredan todas sus propiedades**.
- En general, cuando creamos un objeto nuevo, tenemos la posibilidad de seleccionar cuál será su prototipo.
- Los objetos creados a través de la notación literal están enlazados con *Object.prototype* (un objeto estándar incluido en JavaScript).

Clases vs. Prototipos I

- Los lenguajes orientados a objetos basados en clases como Java o C++, se basan en el concepto de dos entidades distintas: la clase y las instancias.
- **Clase:**
  - Define todas las propiedades que caracteriza a una serie de objetos.
  - La clase es algo abstracto, no como las instancias de los objetos que describe.
  - Por ejemplo, una clase Empleado, puede representar un conjunto concreto de empleados.
- **Una instancia:**
  - Es una representación concreta de esa clase.
  - Por ejemplo, Victoria puede ser una instancia concreta de la clase Empleado, es decir, representa de manera concreta a un empleado.

Clases vs. Prototipos II

  - Una instancia tiene exactamente las mismas propiedades que la clase padre (Ni más, ni menos).
- Un lenguaje basado en prototipos, como JavaScript, no hace esta distinción: simplemente maneja objetos.
- Este tipo de lenguajes tiene la noción de objetos prototipo, objetos usados como platilla para obtener las propiedades iniciales de un objeto.
- Cualquier objeto puede especificar su propias propieades, tanto en el momento que los creamos como en tiempo de ejecución.
- Además, cualquier objeto puede asociarse como prototipo a otro objeto, permitiendo compartir todas sus propiedades.

Definiendo una clase I

- Un lenguaje basado en clases, definimos la clase de manera independiente.
- En esta definición, especificamos los constructores, que son utilizamos para crear las instancias de las clases.
- Un método constructor, puede especificar los valores iniciales de una instancia de una clase, y realizar las operaciones necesarias a la hora de crear el objeto.
- Utilizamos el operador **new**, conjuntamente con el nombre del constructor, para crear nuevas instancias.
- JavaScript sigue un modelo similar, pero no separa la definición de las propiedades del constructor.
- En este caso, definimos una función constructora para crear los objetos con un conjunto inicial de propiedades y valores.

Definiendo una clase II

- Cualquier función de JavaScript puede ser utilizada como constructor.
- Utilizamos el operador **new**, conjuntamente con el nombre de la función constructora, para crear nuevas instancias.

Subclases y herencia

- En un lenguaje basado en clases, es posible crear estructura de clases a través de su definición.
- En esta definición, podemos especificar que la nueva clase es una subclase de una clase que ya existe.
- Esta subclase, hereda todas las propiedades de la superclase, y además puede añadir o modificar las propiedades heredadas.
- JavaScript implementa una herencia que nos permite asociar un objeto prototipo con una función constructora.
- De esta manera, el nuevo objeto hereda todas las propiedades del objeto prototipo.

Creando la herencia I

- Veamos como se implementa esta herencia en JavaScript, a través de un simple ejemplo.
- Queremos implementar la siguiente estructura:
  - Un Empleado se define con las propiedades nombre (cuyo valor por defecto es una cadena vacía), y un departamento (cuyo valor por defecto es "General").
  - Un Director está basado en Empleado. Añade la propiedad informes (cuyo valor por defecto es un array vacío).
  - Un Trabajador está basado también en Empleado. Añade la propiedad proyectos (cuyo valor por defecto es un array vacío).
  - Un Ingeniero está basado en Trabajador. Añade la propiedad maquina (cuyo valor por defecto es una cadena vacía) y sobreescribe la propiedad departamento con el valor "Ingeniería".

Creando la herencia II

- Codificación en JavaScript:

```
1  function Empleado (nombre, departamento) {
2  this.nombre = nombre || "";
3  this.departamento = departamento || "General";
4  }
5
6  function Director (nombre, departamento, informes) {
7    this.base = Empleado;
8    this.base(nombre, departamento);
9    this.informes = informes || [];
10 }
11 Director.prototype = new Empleado;
12
13 function Obrero (nombre, departamento, proyectos) {
14   this.base = Empleado;
15   this.base(nombre, departamento);
16   this.proyectos = proyectos || [];
17 }
18 Obrero.prototype = new Empleado;
19
20 function Ingeniero (nombre, proyectos, maquina) {
21   this.base = Obrero;
22   this.base(nombre, "Ingenieria", proyectos);
23   this.maquina = maquina || "";
24 }
25 Ingeniero.prototype = new Obrero;
```

Crear Objetos de Prototipos I

- El mecanismo general para crear objetos de prototipos que proporciona JavaScript es complejo y seguramente desordenado.
- Sin embargo, siguiendo unas ciertas pautas se puede simplificar de manera significativa.
- Para ilustrarlo, vamos a añadir un método de creación a nuestro objeto.
- El método create crea un nuevo **objeto** que utiliza un objeto antiguo como su prototipo.

Crear Objetos de Prototipos II


```
1  // Shape - superclass
2  function Shape() {
3      this.x = 0;
4      this.y = 0;
5  }
6
7  Shape.prototype.move = function(x, y) {
8      this.x += x;
9      this.y += y;
10     console.info("Shape moved.");
11 };
12
13 // Rectangle - subclass
14 function Rectangle() {
15     Shape.call(this); //call super constructor.
16 }
17
18 Rectangle.prototype = Object.create(Shape.prototype);
19
20 var rect = new Rectangle();
21
22 rect instanceof Rectangle   // true.
23 rect instanceof Shape       // true.
24
25 rect.move(); // Outputs, "Shape moved."
```

Crear Objetos de Prototipos III

- Para los navegadores que no soportan la función create, podemos extender el objeto de JavaScript Object para incluir esta funcionalidad:

```
1  if (typeof Object.create !== 'function') {
2      Object.create = function (o) {
3  var F = function () {};
4  F.prototype = o;
5  return new F();
6      };
7  }
8  var another_stooge = Object.create(stooge);
```

- El prototipo enlazado no se ve afectado por las modificaciones.

Crear Objetos de Prototipos IV

- Si realizamos cambios en un objeto, el objeto prototipo no se ve afectado.

```
1  another_stooge['first-name'] = 'Harry';
2  another_stooge['middle-name'] = 'Moses';
3  another_stooge.nickname = 'Moe';
```

- El enlace de los prototipos es únicamente utilizado cuando accedemos a los datos.
- Si intentamos acceder al valor de una propiedad, y esa propiedad no existe en el objeto, entonces JavaScript va a intentar obtener ese valor del prototipo del objeto.
- Y si ese objeto tampoco dispone de la propiedad, lo intentará obtener de sucesivos prototipos, hasta que finalmente se encuentre con *Object.prototype.*

Crear Objetos de Prototipos V

- Si la propiedad no existe en ninguno de los prototipos, entonces el valor devuelto es *undefined.*
- La relación de prototipos es dinámica.
- Si añadimos una nueva propiedad a un prototipo, entonces esta propiedad estará inmediatamente accesible para el resto de prototipos que estén basados en ese prototipo:

```
1  stooge.profession = 'actor';
2  another_stooge.profession       // 'actor'
```

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Arrays

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

Representación de un Array I

- JavaScript ofrecen una manera muy cómoda para crear y representar arrays.
- Una representación de un array consiste en una pareja de corchetes **([ ])** que contienen cero o más expresiones.
- El primer valor recibe la propiedad de nombre ’0’, la segunda la propiedad de nombre ’1’, y así sucesivamente. Se define de la siguiente manera:

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

- Algunos ejemplos de declaración de arrays:

Representación de un Array II


```
1  var empty = [];
2
3  var numbers = [
4      'zero', 'one', 'two', 'three', 'four',
5      'five', 'six', 'seven', 'eight', 'nine'
6  ];
7
8  empty[1]         // undefined
9  numbers[1]       // 'one'
10
11 empty.length     // 0
12 numbers.length   // 10
```

- Si representasemos el array como un objeto:

```
1  var numbers_object = {
2    '0': 'zero', '1': 'one',    '2': 'two',
3    '3': 'three', '4': 'four', '5': 'five',
4    '6': 'six', '7': 'seven', '8': 'eight',
5    '9': 'nine'
6  };
```

- Se produce un efecto similar. Ambas representaciones contienen 10 propiedades, y estas propiedades tienen exactamente los mismos nombres y valores.

Representación de un Array III

- La diferencia radica en que numbers hereda de *Array.prototype*, mientras que numbers_object lo hace de *Object.prototype*, por lo que numbers hereda una serie de métodos que convierten a numbers en un array.
- En la mayoría de los lenguajes de programación, se requiere que todos los elementos de un array sean del mismo tipo, pero en JavaScript eso no ocurre.
- Un array puede contener una mezcla valores:

```
1  var misc = [
2      'string', 98.6, true, false, null, undefined,
3      ['nested', 'array'], {object: true}, NaN,
4      Infinity
5  ];
6  misc.length // 10
```

Propiedad Length I

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

```
1  var myArray = [];
2  myArray.length              // 0
3
4  myArray[1000000] = true;
5  myArray.length              // 1000001
6
7  // myArray contiene un elemento!var myArray = [];
8  myArray.length              // 0
9
10 myArray[1000000] = true;
11 myArray.length              // 1000001
12
13 // myArray contiene un elemento!
```

Propiedad Length II

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

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

Borrado

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

```
1 delete numbers[2];
2 // numbers es ['zero', 'one', undefined, 'shi', 'go']  , 2da
      posicion vale 'undefined'.
```

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

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

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Funciones y propiedades básicas

- JavaScript incorpora una serie de herramientas y utilidades para el manejo de las variables.
- De esta forma, muchas de las operaciones básicas con las variables, se pueden realizar directamente con las utilidades que ofrece JavaScript.

Funciones útiles para cadenas de textos I

- A continuación se muestran algunas de las funciones más útiles para el manejo de cadenas de texto.
- *length*, calcula la longitud de una cadena de texto (el número de caracteres que la forman):

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

- *+*, se emplea para concatenar varias cadenas de texto:

```
1 var mensaje1 = "Hola";
2 var mensaje2 = " Mundo";
3 var mensaje = mensaje1 + mensaje2; // mensaje = "Hola Mundo"
```

- Además del operador +, también se puede utilizar la función *concat()*:

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

Funciones útiles para cadenas de textos II

- Los espacios en blanco se pueden añadir al final o al principio de las cadenas y también se pueden indicar forma explícita:

```
1 var mensaje1 = "Hola";
2 var mensaje2 = "Mundo";
3 var mensaje = mensaje1 + " " + mensaje2; // mensaje = "Hola Mundo"
```

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

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

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

Funciones útiles para cadenas de textos III

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

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

```
1 var mensaje = "Hola";
2 var letra = mensaje.charAt(0); // letra = H
3 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 la cadena no contiene el carácter, la función devuelve el valor -1:

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

Funciones útiles para cadenas de textos IV

- *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:

```
1 var mensaje = "Hola";
2 var posicion = mensaje.lastIndexOf('a'); // posicion = 3
3 posicion = mensaje.lastIndexOf('b');     // posicion = -1
4 //  La funcion lastIndexOf() comienza su busqueda desde el final de
        la cadena hacia el principio, aunque la posicion 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:

Funciones útiles para cadenas de textos V

```
1 var mensaje = "Hola Mundo";
2 var porcion = mensaje.substring(2); // porcion = "la Mundo"
3 porcion = mensaje.substring(5);     // porcion = "Mundo"
4 porcion = mensaje.substring(7);     // porcion = "ndo"
5 //Si se indica un inicio negativo, se devuelve la misma cadena
      original.
```

- *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:

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

Funciones útiles para arrays I

- A continuación se muestran algunas de las funciones más útiles para el manejo de arrays:
- *length*, calcula el número de elementos de un array:

```
1 var vocales = ["a", "e", "i", "o", "u"];
2 var numeroVocales = vocales.length; // numeroVocales = 5
```

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

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

Funciones útiles para arrays II


```
1 var array = ["hola", "mundo"];
2 var mensaje = array.join(""); // mensaje = "holamundo"
3 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:

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

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

Funciones útiles para arrays III

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

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

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

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

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

Funciones útiles para números I

- A continuación se muestran algunas de las funciones y propiedades más útiles para el manejo de números.
- *NaN*, (del inglés, "Not a Number") JavaScript emplea el valor NaN para indicar un valor numérico no definido (por ejemplo, la división 0/0):

```
1 var numero1 = 0;
2 var numero2 = 0;
3 alert(numero1/numero2); // se muestra el valor NaN
```

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

```
1 var numero1 = 0;
2 var numero2 = 0;
3 if(isNaN(numero1/numero2)) {
4 alert("La division no esta definida para los numeros indicados");
5 }
6 else {
7 alert("La division es igual a => " + numero1/numero2);
8 }
```

Funciones útiles para números II

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

```
1 var numero1 = 10;
2 var numero2 = 0;
3 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:

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

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

JSON

- JavaScript Object Notation (JSON) es un formato de intercambio de datos muy ligero y está basdo en la notación para la representación de objetos.
- A pesar de estar basado en JavaScript, es independiente del lenguaje.
- Puede ser utilizado para intercambiar datos entre programas escritos en lenguajes totalmente diferentes.
- Es un formato de texto, por lo que puede ser leído por máquinas y humanos, he implementado de una manera muy sencilla.
- Para acceder a toda la información sobre JSON, acceder a *http://www.JSON.org/.*

Sintaxis JSON I

- JSON define seis tipos de valores: *objects, arrays, strings, numbers, booleans y el valor especial null.*
- Los espacios (espacios en blanco, tabuladores, retornos de carro y nueva línea) pueden introducirse antes o después de cualquier valor, sin afectar a los valores representados.
- Esto hace que un texto JSON sea mucho más fácil de leer por humanos.
- Un objeto JSON es un contenedor, no ordenado de parejas clave/valor.
- Una clave puede ser un string, y un valor puede ser un valor JSON (tanto un array como un objeto).
- Los objetos JSON se pueden anidar hasta cualquier profundidad.

Sintaxis JSON II

- Un array JSON es una secuencia ordenada de valores, donde un valor puede ser un valor JSON (tanto un array como un objeto).
- La gran mayoría de lenguajes incluyen características para trabajar de manera cómoda con valores JSON en ambos sentidos: partiendo de un objeto u array para convertirlo a una cadena de caracteres, o a partir de una cadena de caracteres, obtener los valores JSON.
- La sintaxis de los valores JSON es la siguiente:

```
1   "firstName":"John"
```

- Los nombres JSON requieren comillas dobles.
- **Objetos JSON:**
- Los objetos JSON se escriben dentro de llaves.

Sintaxis JSON III

- Al igual que JavaScript, los objetos JSON pueden contener múltiples pares de nombre / valores:

```
{"firstName":"John", "lastName":"Doe"}
```

- **Los Arrays en JSON:**
- Los arrays en JSON se escriben entre corchetes.
- Al igual que JavaScript, una array JSON puede contener varios objetos:

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter","lastName":"Jones"}
]
```

- En el ejemplo anterior, el objeto "employees" es un array que contiene tres objetos.
- Cada objeto es un registro de una persona (con un nombre y un apellido).

Sintaxis JSON IV

- **Sintaxis JSON utilizada en JavaScript:**
- Debido a que la sintaxis JSON se deriva de la notación de objetos en JavaScript, es muy sencillo implementarlo.
- Con JavaScript se puede crear un array de objetos y asignar los datos a la misma, así:

```
1  var employees = [
2    {"firstName":"John", "lastName":"Doe"},
3    {"firstName":"Anna", "lastName":"Smith"},
4    {"firstName":"Peter","lastName": "Jones"}
5  ];
```

- Se puede acceder a la primera entrada del array del objeto JavaScript de la siguiente manera:

```
1  // returns John Doe
2  employees[0].firstName + " " + employees[0].lastName;
```

- También se puede acceder de esta otra manera:

```
1  // returns John Doe
2  employees[0]["firstName"] + " " + employees[0]["lastName"];
```

Outline

1 JavaScript
Introducción
Variables
Operadores
Estructuras de control
Funciones
Objetos
Herencia
Arrays
Utilidades
JSON
DOM
Eventos

Document Object Model I

- 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 XHTML 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.

Árbol de nodos I

- 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, , 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.

Árbol de nodos II

- Una página xHTML 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.

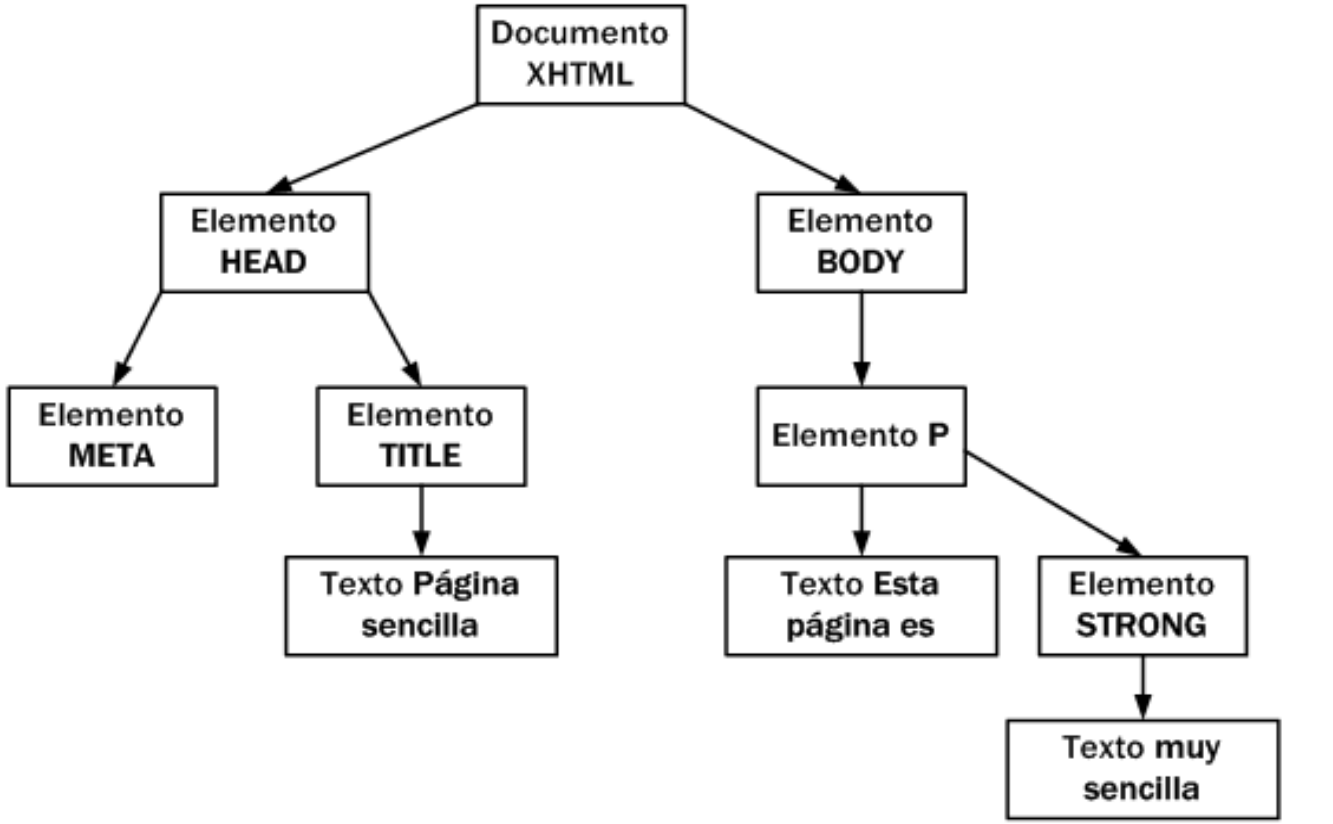
Árbol de nodos III

- DOM transforma todos los documentos XHTML 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 XHTML sencilla:

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
       http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2  <html xmlns="http://www.w3.org/1999/xhtml">
3  <head>
4  <meta http-equiv="Content-Type" content="text/html;
       charset=iso-8859-1" />
5  <title>Pagina sencilla</title>
6  </head>
7
8  <body>
9  <p>Esta pagina es <strong>muy sencilla</strong></p>
10 </body>
11 </html>
```

Árbol de nodos IV

- Se transforma en el siguiente árbol de nodos:



- ***Figura 1.*** *Árbol de nodos generado automáticamente por DOM a partir del código XHTML de la página.*

Árbol de nodos V

- 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 XHTML siempre es la misma: un nodo de tipo especial denominado **"Documento".**
- A partir de ese nodo raíz, cada etiqueta XHTML 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.**

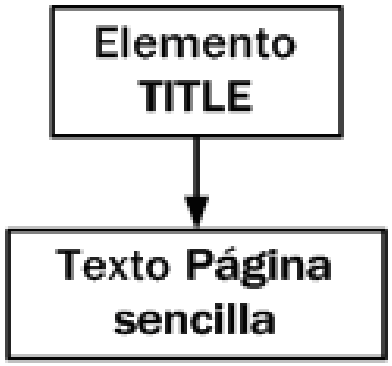
Árbol de nodos VI

- A partir de esta derivación inicial, cada etiqueta XHTML se transforma en un nodo que deriva del nodo correspondiente a su "etiqueta padre".
- La transformación de las etiquetas XHTML habituales genera dos nodos: el primero es el nodo de tipo "Elemento" (correspondiente a la propia etiqueta XHTML) y el segundo es un nodo de tipo "Texto" que contiene el texto encerrado por esa etiqueta XHTML.
- Así, la siguiente etiqueta XHTML:

```
1 | <title>Pagina sencilla</title>
```

Árbol de nodos VII

- Genera los siguientes dos nodos:



  - ***Figura 2.*** *Nodos generados automáticamente por DOM para una etiqueta XHTML sencilla.*

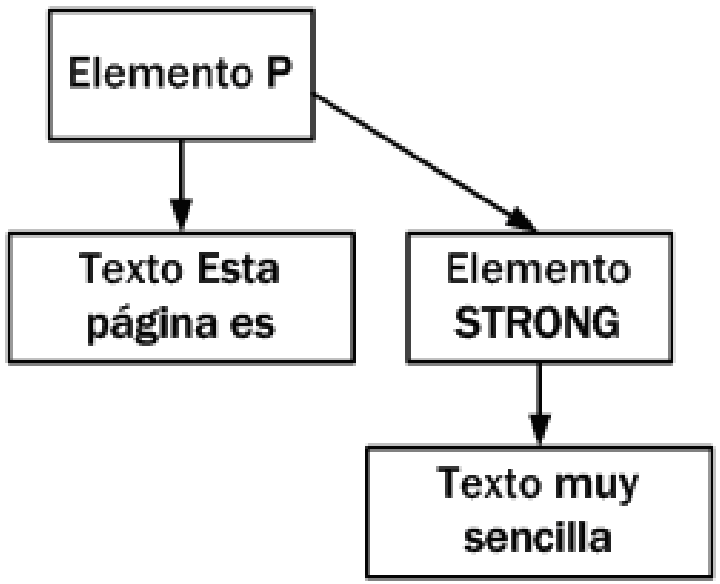
- De la misma forma, la siguiente etiqueta XHTML:

```
1 | <p>Esta pagina 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>.

Árbol de nodos VIII

- Como el contenido de <p> incluye en su interior otra etiqueta XHTML, 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 .



Árbol de nodos IX

- ***Figura 3.*** *Nodos generados automáticamente por DOM para una etiqueta XHTML con otras etiquetas XHTML en su interior.*

- La transformación automática de la página en un árbol de nodos siempre sigue las mismas reglas:
  - Las etiquetas XHTML 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 XHTML se encuentra dentro de otra, se sigue el mismo procedimiento anterior, pero los nodos generados serán nodos hijo de su etiqueta padre.

Árbol de nodos X

- Como se puede suponer, las páginas XHTML habituales producen árboles con miles de nodos.
- Sin embargo, 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.

Tipo de nodos

- La especificación completa de DOM define 12 tipos de nodos.
- Las páginas XHTML 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 XHTML. 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 XHTML, es decir, uno por cada par atributo=valor.
  - **Text**, nodo que contiene el texto encerrado por una etiqueta XHTML.
  - **Comment**, representa los comentarios incluidos en la página XHTML.

Acceso directo a los nodos I

- 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 directo a los nodos II

- **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.
- **Acceso directo:**
  - 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, vamos a presentar solo las funciones 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 XHTML se cargue por completo.

Acceso directo a los nodos III

- **getElementsByTagName():**
  - La función getElementsByTagName(nombreEtiqueta) obtiene todos los elementos de la página XHTML 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 XHTML:

```
1 | 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.

Acceso directo a los nodos IV

- 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:

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

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

```
1  for(var i=0; i<parrafos.length; i++) {
2    var parrafo = parrafos[i];
3  }
```

- 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:

Acceso directo a los nodos V

```
1  var parrafos = document.getElementsByTagName("p");
2  var primerParrafo = parrafos[0];
3  var enlaces = primerParrafo.getElementsByTagName("a");
```

- **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:

```
1  var parrafoEspecial = document.getElementsByName("especial")
       ;
2
3  <p name="prueba">...</p>
4  <p name="especial">...</p>
5  <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.

Acceso directo a los nodos VI

- 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.

- **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 XHTML 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.

Acceso directo a los nodos VII


```
1  var cabecera = document.getElementById("cabecera");
2
3  <div id="cabecera">
4    <a href="/" id="logo">...</a>
5  </div>
```

Creación y eliminación de nodos I

- 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.
- **Creación de elementos XHTML simples:**
  - Como se ha visto, un elemento XHTML sencillo, como por ejemplo un párrafo, genera dos nodos:
    - El primer nodo es de tipo *Element* y representa la etiqueta **<p>**.
    - El segundo nodo es de tipo *Text* y representa el contenido **textual** de la etiqueta <p>.

Creación y eliminación de nodos II

- Por este motivo, crear y añadir a la página un nuevo elemento XHTML sencillo consta de cuatro pasos diferentes:
  - Creación de un nodo de tipo Element que represente al elemento.
  - Creación de un nodo de tipo Text que represente el contenido del elemento.
  - Añadir el nodo Text como nodo hijo del nodo Element.
  - 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 XHTML, es necesario incluir el siguiente código JavaScript:

Creación y eliminación de nodos III

```
1  // Crear nodo de tipo Element
2  var parrafo = document.createElement("p");
3
4  // Crear nodo de tipo Text
5  var contenido = document.createTextNode("Hola Mundo!");
6
7  // Anadir el nodo Text como hijo del nodo Element
8  parrafo.appendChild(contenido);
9
10 // Anadir el nodo Element como hijo de la pagina
11 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 XHTML cuya etiqueta se pasa como parámetro.
  - **createTextNode(contenido):** crea un nodo de tipo Text que almacena el contenido textual de los elementos XHTML.

Creación y eliminación de nodos IV

- **nodoPadre.appendChild(nodoHijo):** añade un nodo como hijo de otro nodo. Se debe utilizar al menos dos veces con los nodos habituales, tal y como lo indica nuestro ejemplo anterior.

- **Eliminación de nodos:**
  - Afortunadamente, eliminar un nodo del árbol DOM de la página es mucho más sencillo que añadirlo.
  - En este caso, solamente es necesario utilizar la función **removeChild()**:

```
1  var parrafo = document.getElementById("provisional");
2  parrafo.parentNode.removeChild(parrafo);
3
4  <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.

Creación y eliminación de nodos V

- 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 XHTML 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.

Acceso directo a los atributos XHTML I

- Una vez que se ha accedido a un nodo, el siguiente paso natural consiste en acceder y/o modificar sus atributos y propiedades.
- Mediante DOM, es posible acceder de forma sencilla a todos los atributos XHTML y todas las propiedades **CSS** de cualquier elemento de la página.
- Los atributos XHTML de los elementos de la página se transforman automáticamente en propiedades de los nodos.
- Para acceder a su valor, simplemente se indica el nombre del atributo XHTML detrás del nombre del nodo.

Acceso directo a los atributos XHTML II

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

```
1 var enlace = document.getElementById("enlace");
2 alert(enlace.href); // muestra http://www...com
3
4 <a id="enlace" href="http://www...com">Enlace</a>
```

- En el ejemplo anterior, se obtiene el nodo DOM que representa el enlace mediante la función **document.getElementById()**.
- A continuación, se obtiene el atributo **href** del enlace mediante *enlace.href*.
- Para obtener por ejemplo el atributo id, se utilizaría enlace.id.
- Las propiedades **CSS** no son tan fáciles de obtener como los atributos XHTML.

Acceso directo a los atributos XHTML III

- Para obtener el valor de cualquier propiedad CSS del nodo, se debe utilizar el atributo **style**.
- El siguiente ejemplo obtiene el valor de la propiedad margin de la imagen:

```
1  var imagen = document.getElementById("imagen");
2  alert(imagen.style.margin);
3
4  <img id="imagen" style="margin:0; border:0;" src="logo.png" />
```

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

```
1  var parrafo = document.getElementById("parrafo");
2  alert(parrafo.style.fontWeight);  // muestra "bold"
3
4  <p id="parrafo" style="font-weight: bold;">...</p>
```

Acceso directo a los atributos XHTML IV

- 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 XHTML que no tiene el mismo nombre en XHTML 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 XHTML.

Acceso directo a los atributos XHTML V

- En su lugar, DOM utiliza el nombre className para acceder al atributo class de XHTML:

```
1  var parrafo = document.getElementById("parrafo");
2  alert(parrafo.class); // muestra "undefined"
3  alert(parrafo.className); // muestra "normal"
4
5  <p id="parrafo" class="normal">...</p>
```

Outline

JavaScript

1 JavaScript
- Introducción
- Variables
- Operadores
- Estructuras de control
- Funciones
- Objetos
- Herencia
- Arrays
- Utilidades
- JSON
- DOM
- Eventos

Eventos I

- En la programación tradicional, las aplicaciones se ejecutan secuencialmente de principio a fin para producir sus resultados.
- Sin embargo, en la actualidad el modelo predominante es el de la **programación basada en eventos.**
- Los scripts y programas esperan sin realizar ninguna tarea hasta que se produzca un evento.
- Una vez producido, ejecutan alguna tarea asociada a la aparición de ese evento y cuando concluye, el script o programa vuelve al estado de espera.
- JavaScript permite realizar scripts con ambos métodos de programación: *secuencial y basada en eventos.*
- Los eventos de JavaScript permiten la interacción entre las aplicaciones JavaScript y los usuarios.

Eventos II

- Cada vez que se pulsa un botón, se produce un evento, o cuando se pulsa una tecla, también se produce un evento.
- No obstante, para que se produzca un evento no es obligatorio que intervenga el usuario, ya que por ejemplo, cada vez que se carga una página, también se produce un evento.
- El nivel 1 de DOM no incluye especificaciones relativas a los eventos JavaScript.
- El nivel 2 de DOM incluye ciertos aspectos relacionados con los eventos y el nivel 3 de DOM incluye la especificación completa de los eventos de JavaScript.

Eventos III

- Desafortunadamente, la especificación de nivel 3 de DOM se publicó en el año 2004, más de doce años después de que los primeros navegadores incluyeran los eventos.
- Por este motivo, muchas de las propiedades y métodos actuales relacionados con los eventos son incompatibles con los de DOM.
- De hecho, navegadores como Internet Explorer tratan los eventos siguiendo su propio modelo incompatible con el estándar.
- El modelo simple de eventos se introdujo en la versión 4 del estándar HTML y se considera parte del nivel más básico de DOM.

Eventos IV

- Aunque sus características son limitadas, es el único modelo que es compatible con todos los navegadores y por tanto, el único que permite crear aplicaciones que funcionan de la misma manera en todos los navegadores.

Tipos de eventos I

- Cada elemento XHTML tiene definida 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 XHTML y un mismo elemento XHTML puede tener asociados diferentes eventos.
- El nombre de los eventos se construye mediante el prefijo **on**, seguido del nombre en inglés de la acción asociada al evento.
- Así, el evento de pulsar un elemento con el ratón se denomina **onclick** y el evento asociado a la acción de mover el ratón se denomina **onmousemove**.

Tipos de eventos II

- A continuación mostramos un resumen de los eventos más importantes definidos por JavaScript:
- **onblur**
  - Descripción: Un elemento pierde el foco.
  - Elementos: <button>, <input>, <label>, <select>, <textarea>, <body>.
- **onchange**
  - Descripción: Un elemento ha sido modificado.
  - Elementos: <input>, <select>, <textarea>.
- **onclick**
  - Descripción: Pulsar y soltar el ratón.
  - Elementos: Todos los elementos.
- **ondblclick**
  - Descripción: Pulsar dos veces seguidas con el ratón.
  - Elementos: Todos los elementos.

Tipos de eventos III

- **onfocus**
  - Descripción: Un elemento pierde el foco.
  - Elementos: <button>, <input>, <label>, <select>, <textarea>, <body>
- **onkeydown**
  - Descripción: Pulsar una tecla y no soltarla.
  - Elementos: Elementos de formulario y <body>.
- **onkeypress**
  - Descripción: Pulsar una tecla.
  - Elementos: Elementos de formulario y <body>.
- **onkeyup**
  - Descripción: Soltar una tecla pulsada.
  - Elementos: Elementos de formulario y <body>.
- **onload**
  - Descripción: Página cargada completamente.
  - Elementos: <body>.

Tipos de eventos IV

- **onmousedown**
  - Descripción: Pulsar un botón del ratón y no soltarlo.
  - Elementos: Todos los elementos.
- **onmousemove**
  - Descripción: Mover el ratón.
  - Elementos: Todos los elementos.
- **onmouseout**
  - Descripción: El ratón "sale" del elemento.
  - Elementos: Todos los elementos.
- **onmouseover**
  - Descripción: El ratón "entra" en el elemento.
  - Elementos: Todos los elementos.
- **onmouseup**
  - Descripción: Soltar el botón del ratón.
  - Elementos: Todos los elementos.

Tipos de eventos V

- **onreset**
  - Descripción: Inicializar el formulario.
  - Elementos: <form>.
- **onresize**
  - Descripción: Modificar el tamaño de la ventana.
  - Elementos: <body>.
- **onselect**
  - Descripción: Seleccionar un texto.
  - Elementos: <input>, <textarea>.
- **onsubmit**
  - Descripción: Enviar el formulario.
  - Elementos: <form>.
- **onunload**
  - Descripción: Se abandona la página, por ejemplo al cerrar el navegador.
  - Elementos: <body>.

Tipos de eventos VI

- 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.
- 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.

Manejadores de eventos I

- Un evento de JavaScript por sí mismo carece de utilidad.
- 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 (event handlers en inglés) y como JavaScript es un lenguaje muy flexible, existen varias formas diferentes de indicar los manejadores:
  - Manejadores como atributos de los elementos XHTML.
  - Manejadores como funciones JavaScript externas.
  - Manejadores "semánticos".

Manejadores de eventos II

- **MANEJADORES COMO ATRIBUTOS XHTML:**
  - Se trata del método más sencillo y a la vez menos profesional de indicar el código JavaScript que se debe ejecutar cuando se produzca un evento.
  - En este caso, el código se incluye en un atributo del propio elemento XHTML.
  - En el siguiente ejemplo, se quiere mostrar un mensaje cuando el usuario pulse con el ratón sobre un botón:

```
1  <input type="button" value="Pulsar" onclick="alert('Gracias
       por pulsar');" />
```

  - En este método, se definen atributos XHTML con el mismo nombre que los eventos que se quieren manejar.
  - El ejemplo anterior sólo quiere controlar el evento de pinchar con el ratón, cuyo nombre es onclick.
  - Así, el elemento XHTML para el que se quiere definir este evento, debe incluir un atributo llamado onclick.

Manejadores de eventos III

- El contenido del atributo es una cadena de texto que contiene todas las instrucciones JavaScript que se ejecutan cuando se produce el evento.
- En este caso, el código JavaScript es muy sencillo (alert('Gracias por pulsar');), ya que solamente se trata de mostrar un mensaje.
- En este otro ejemplo, cuando el usuario pulsa sobre el elemento se muestra un mensaje y cuando el usuario pasa el ratón por encima del elemento, se muestra otro mensaje:

```
1  <div onclick="console.log('Has pulsado con el raton');"
       onmouseover="console.log('Acabas de pasar el raton por
       encima');">
2    Puedes pulsar sobre este elemento o simplemente pasar el
       raton por encima
3  </div>
4  //usamos console.log(), para poder mostrar un mensaje por
       consola.
```

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

Manejadores de eventos IV


```
1  <body onload="console.log('La pagina se ha cargado
       completamente');">
2  ...
3  </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, como se vio en el capítulo de DOM, las funciones que permiten acceder y manipular los nodos del árbol DOM solamente están disponibles cuando la página se ha cargado completamente.

Manejadores de eventos V

- **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 XHTML que ha provocado el evento.
  - Esta variable es muy útil para ejemplos como el siguiente:
  - Cuando el usuario pasa el ratón por encima del , el color del borde se muestra de color negro. Cuando el ratón sale del , se vuelve a mostrar el borde con el color gris claro original.

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

Manejadores de eventos VI

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

```
1  <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';">
2    Seccion de contenidos...
3  </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 XHTML que ha provocado el evento.
- Así, el ejemplo anterior se puede reescribir de la siguiente manera:

Manejadores de eventos VII


```
1  <div id="contenidos" style="width:150px; height:60px;
       border:thin solid silver" onmouseover="
       this.style.borderColor='black';" onmouseout="
       this.style.borderColor='silver';">
2    Seccion de contenidos...
3  </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 .

Manejadores de eventos VIII

- **MANEJADORES DE EVENTOS COMO FUNCIONES EXTERNAS:**
  - La definición de manejadores de eventos en los atributos XHTML es un método sencillo pero poco aconsejable para 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.
  - Cuando el código de la función manejadora es más complejo, como por ejemplo la validación de un formulario, es aconsejable agrupar todo el código JavaScript en una función externa que se invoca desde el código XHTML cuando se produce el evento.

Manejadores de eventos IX

- De esta forma, el siguiente ejemplo:

```
1  <input type="button" value="Pulsar" onclick="console.log('
       Gracias por pulsar');" />
```

- Se puede transformar en:

```
1  function muestraMensaje() {
2   console.log('Gracias por pulsar');
3  }
4  <input type="button" value="Pulsar" onclick="muestraMensaje
       ()" />
```

- En las funciones externas no es posible utilizar la variable this de la misma forma que en los manejadores insertados en los atributos XHTML.
- Por tanto, es necesario pasar la variable this como parámetro a la función manejadora:

Manejadores de eventos X

```
1   function resalta(elemento) {
2     switch(elemento.style.borderColor) {
3       case 'silver':
4       case 'silver silver silver silver':
5       case '#c0c0c0':
6   elemento.style.borderColor = 'black';
7   break;
8       case 'black':
9       case 'black black black black':
10      case '#000000':
11  elemento.style.borderColor = 'silver';
12  break;
13      }
14    }
15
16    <div style="padding: .2em; width: 150px; height: 60px;
          border: thin solid silver" onmouseover="resalta(this)"
          onmouseout="resalta(this)">
17      Seccion de contenidos...
18    </div>
```

- En el ejemplo anterior, a la función externa se le pasa el parámetro **this**, que dentro de la función se denomina *elemento.*
- Al pasar this como parámetro, es posible acceder de forma directa desde la función externa a las propiedades del elemento que ha provocado el evento.

Manejadores de eventos XI

- **MANEJADORES DE EVENTOS SEMÁNTICOS:**
  - Utilizar los atributos XHTML o las funciones externas para añadir manejadores de eventos tiene un grave inconveniente: "ensucian" el código XHTML de la página.
  - Como es conocido, al crear páginas web se recomienda separar los contenidos (XHTML) de la presentación (CSS).
  - En lo posible, también se recomienda separar los contenidos (XHTML) de la programación (JavaScript).
  - Mezclar JavaScript y XHTML complica excesivamente el código fuente de la página, dificulta su mantenimiento y reduce la semántica del documento final producido.
  - Afortunadamente, existe un método alternativo para definir los manejadores de eventos de JavaScript.

Manejadores de eventos XII

- Esta técnica consiste en asignar las funciones externas mediante las propiedades DOM de los elementos XHTML. Así, el siguiente ejemplo:

```
1  <input id="test" type="button" value="Pulsar" onclick="
       console.log('Gracias por pulsar');" />
2  Se puede transformar en:
3  function muestraMensaje() {
4    console.log('Gracias por pulsar');
5  }
6  document.getElementById("test").onclick = muestraMensaje;
7  <input id="test" type="button" value="Pulsar" />
```

- El código XHTML resultante es muy "limpio", ya que no se mezcla con el código JavaScript.
- La técnica de los manejadores semánticos consiste en:
  - Asignar un identificador único al elemento XHTML mediante el atributo id.
  - Crear una función de JavaScript encargada de manejar el evento.
  - Asignar la función a un evento concreto del elemento XHTML mediante DOM.

Manejadores de eventos XIII

- Otra ventaja adicional de esta técnica es que las funciones externas pueden utilizar la variable **this** referida al elemento que original el evento.
- Asignar la función manejadora mediante DOM es un proceso que requiere una explicación detallada.
- En primer lugar, se obtiene la referencia del elemento al que se va a asignar el manejador:

```
1 | document.getElementById("pulsable");
```

- A continuación, se asigna la función externa al evento deseado mediante una propiedad del elemento con el mismo nombre del evento:

```
1 | document.getElementById("pulsable").onclick = ...
```

- Por último, se asigna la función externa. Como ya se ha comentado en capítulos anteriores, 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:

Manejadores de eventos XIV


```
1  document.getElementById("pulsable").onclick =
       muestraMensaje;
```

- Si se añaden los paréntesis al final, en realidad se está invocando la función y asignando el valor devuelto por la función al evento onclick de elemento.
- El único inconveniente de este método es que los manejadores se asignan mediante las funciones DOM, que solamente se pueden utilizar después de que la página se ha cargado completamente.
- De esta forma, para que la asignación de los manejadores no resulte errónea, es necesario asegurarse de que la página ya se ha cargado.
- 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**:

```
1  window.onload = function() {
2   document.getElementById("pulsable").onclick =
        muestraMensaje;
3  }
```

Manejadores de eventos XV

- La técnica anterior utiliza una función anónima para asignar algunas instrucciones al evento onload de la página (en este caso se ha establecido mediante el objeto window).
- De esta forma, para asegurar que cierto código se va a ejecutar después de que la página se haya cargado, sólo es necesario incluirlo en el interior de la siguiente construcción:

```
1  window.onload = function() {
2    ...
3  }
```

El flujo de eventos I

- Además de los eventos básicos que se han visto, los navegadores incluyen un mecanismo relacionado llamado flujo de eventos o **"event flow"**.
- El flujo de eventos permite que varios elementos diferentes puedan responder a un mismo evento.
- Si en una página HTML se define un elemento con un botón en su interior, cuando el usuario pulsa sobre el botón, el navegador permite asignar una función de respuesta al botón, otra función de respuesta al que lo contiene y otra función de respuesta a la página completa.
- De esta forma, un solo evento (la pulsación de un botón) provoca la respuesta de tres elementos de la página (incluyendo la propia página).

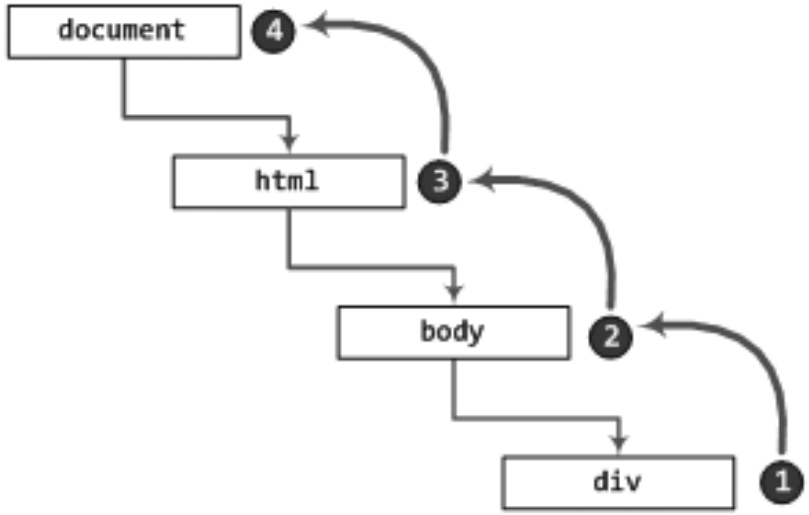
El flujo de eventos II

- El orden en el que se ejecutan los eventos asignados a cada elemento de la página es lo que constituye el flujo de eventos.
- Además, existen muchas diferencias en el flujo de eventos de cada navegador.
- **EVENT BUBBLING:**
  - En este modelo de flujo de eventos, el orden que se sigue es desde el elemento más específico hasta el elemento menos específico.
  - En los próximos ejemplos se emplea la siguiente página HTML:

```
1  <html onclick="procesaEvento()">
2    <head><title>Ejemplo de flujo de eventos</title></head>
3    <body onclick="procesaEvento()">
4      <div onclick="procesaEvento()">Pulsa aqui</div>
5    </body>
6  </html>
```

El flujo de eventos III

- Cuando se pulsa sobre el texto "Pulsa aquí" que se encuentra dentro del , se ejecutan los siguientes eventos en el orden que muestra el siguiente esquema:



  - **Figura 3.** Esquema del funcionamiento del "event bubbling".
- El primer evento que se tiene en cuenta es el generado por el que contiene el mensaje.
- A continuación el navegador recorre los ascendentes del elemento hasta que alcanza el nivel superior, que es el elemento document.

El flujo de eventos IV

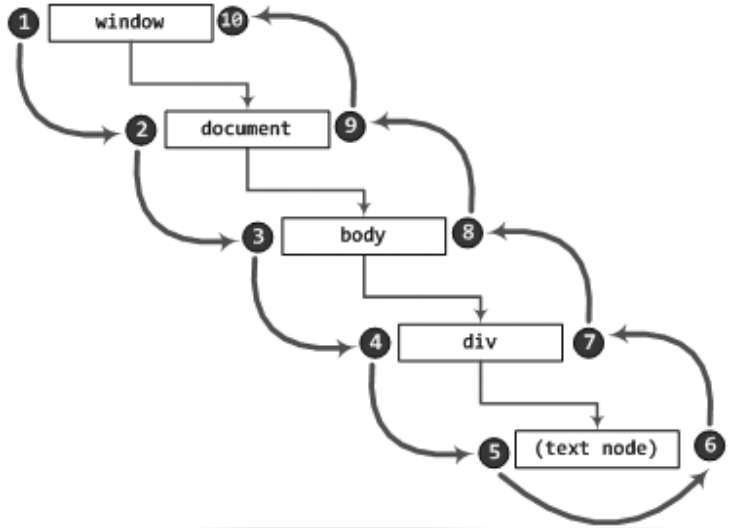
- **EVENT CAPTURING:**
  - En ese otro modelo, el flujo de eventos se define desde el elemento menos específico hasta el elemento más específico.
  - En otras palabras, el mecanismo definido es justamente el contrario al "event bubbling".
  - Este modelo lo utilizaba el desaparecido navegador Netscape Navigator 4.0.

El flujo de eventos V

- **EVENTOS DOM:**
  - El flujo de eventos definido en la especificación DOM soporta tanto el bubbling como el capturing, pero el "event capturing" se ejecuta en primer lugar.
  - Los dos flujos de eventos recorren todos los objetos DOM desde el objeto document hasta el elemento más específico y viceversa.
  - Además, la mayoría de navegadores que implementan los estándares, continúan el flujo hasta el objeto *window*.

El flujo de eventos VI

- El flujo de eventos DOM del ejemplo anterior se muestra a continuación:



  - **Figura 4.** Esquema del flujo de eventos del modelo DOM.
- El elemento más específico del flujo de eventos no es el que desencadena la ejecución de los eventos, sino el nodo de tipo TextNode que contiene el .
- El hecho de combinar los dos flujos de eventos, provoca que el nodo más específico pueda ejecutar dos eventos de forma consecutiva.

HANDLERS Y LISTENERS I

- En las secciones anteriores se introdujo el concepto de "event handler" o manejador de eventos, que son las funciones que responden a los eventos que se producen.
- Además, se vieron tres formas de definir los manejadores de eventos para el modelo básico de eventos:
  - Código JavaScript dentro de un atributo del propio elemento HTML.
  - Definición del evento en el propio elemento HTML pero el manejador es una función externa.
  - Manejadores semánticos asignados mediante DOM sin necesidad de modificar el código HTML de la página.
- Cualquiera de estos tres modelos funciona correctamente en todos los navegadores disponibles en la actualidad.

HANDLERS Y LISTENERS II

- Las diferencias entre navegadores surgen cuando se define más de un manejador de eventos para un mismo evento de un elemento.
- La forma de asignar y "desasignar" manejadores múltiples depende completamente del navegador utilizado.
- **MANEJADORES DE EVENTOS DE DOM:**
  - La especificación DOM define otros dos métodos similares a los disponibles para Internet Explorer y denominados addEventListener() y removeEventListener() para asociar y desasociar manejadores de eventos.
  - La principal diferencia entre estos métodos y los anteriores es que en este caso se requieren tres parámetros: el nombre del "event listener", una referencia a la función encargada de procesar el evento y el tipo de flujo de eventos al que se aplica.

HANDLERS Y LISTENERS III

- El primer argumento no es el nombre completo del evento como sucede en el modelo de Internet Explorer, sino que se debe eliminar el prefijo **on**.
- En otras palabras, si en Internet Explorer se utilizaba el nombre onclick, ahora se debe utilizar click.
- Si el tercer parámetro es true, el manejador se emplea en la fase de capture.
- Si el tercer parámetro es false, el manejador se asocia a la fase de bubbling.
- A continuación, se muestra un ejemplo empleando los métodos definidos por DOM:

```
1  function muestraMensaje() {
2    console.log("Has pulsado el raton");
3  }
4  var elDiv = document.getElementById("div_principal");
5  elDiv.addEventListener("click", muestraMensaje, false);
6
7  elDiv.removeEventListener("click", muestraMensaje, false);
```

EL OBJETO EVENT I

- Cuando se produce un evento, no es suficiente con asignarle una función responsable de procesar ese evento.
- Normalmente, la función que procesa el evento necesita información relativa al evento producido: la tecla que se ha pulsado, la posición del ratón, el elemento que ha producido el evento, etc.
- El objeto **event** es el mecanismo definido por los navegadores para proporcionar toda esa información.
- Se trata de un objeto que se crea automáticamente cuando se produce un evento y que se destruye de forma automática cuando se han ejecutado todas las funciones asignadas al evento.

EL OBJETO EVENT II

- El estándar DOM especifica que el objeto event es el único parámetro que se debe pasar a las funciones encargadas de procesar los eventos.
- Por tanto, en los navegadores que siguen los estándares, se puede acceder al objeto event a través del array de los argumentos de la función:

```
1  elDiv.onclick = function() {
2    var elEvento = arguments[0];
3  }
```

- También es posible indicar el nombre argumento de forma explícita:

```
1  elDiv.onclick = function(event) {
2    var elEvento =event;
3  }
```

- Por otra parte, Internet explorer considera que este objeto forma parte del objeto **window**.

EL OBJETO EVENT III

- Es decir, en los navegadores tipo Internet Explorer, el objeto event se obtiene directamente mediante:

```
1 |   var evento = window.event;
```

- El funcionamiento de los navegadores que siguen los estándares puede parecer "mágico", ya que en la declaración de la función se indica que tiene un parámetro, pero en la aplicación no se pasa ningún parámetro a esa función.
- En realidad, los navegadores que siguen los estándares crean automáticamente ese parámetro y lo pasan siempre a la función encargada de manejar el evento.
- Una vez obtenido el objeto event, ya se puede acceder a toda la información relacionada con el evento, que depende del tipo de evento producido.

EL OBJETO EVENT IV

- **PROPIEDADES Y MÉTODOS:**
  - A pesar de que el mecanismo definido por los navegadores para el objeto event es similar, existen numerosas diferencias en cuanto las propiedades y métodos del objeto.
  - Existen muchas propiedades definidas para el objeto **event**, como por ejemplo la propiedad **type**, que nos indica el tipo de evento producido:

```
1   function resalta(elEvento) {
2     var evento = elEvento || window.event;
3     switch(evento.type) {
4     case 'mouseover':
5       this.style.borderColor = 'black';
6       break;
7     case 'mouseout':
8       this.style.borderColor = 'silver';
9       break;
10    }
11  }
12  window.onload = function() {
13    document.getElementById("seccion").onmouseover = resalta;
14    document.getElementById("seccion").onmouseout = resalta;
15  }
16
```

EL OBJETO EVENT V


```
17    <div id="seccion" style="width:150px; height:60px;
            border:thin solid silver">
18      Seccion de contenidos...
19    </div>
```

- La propiedad type devuelve el tipo de evento producido, que es igual al nombre del evento pero sin el prefijo on.
- Como ya hemos comentado existen varias propiedades o métodos definidos para el objeto event, tales como **clientX** y **clientY**, útil para obtener las coordenadas del mouse, o **keyCode** que indica el código numérico de la tecla pulsada.