

# Introducción a JavaScript

Luis Valencia Cabrera (lvalencia@us.es),  
Víctor Ramos González (vramos1@us.es)

Departamento de Ciencias de la Computación e Inteligencia Artificial  
Universidad de Sevilla

15-03-2023, Bases de Datos

# Índice

- 1 **Introducción**
- 2 Variables y tipos
- 3 Instrucciones y funciones
- 4 DOM y eventos
- 5 Miscelánea

- JavaScript (JS) es un lenguaje de programación que se utiliza principalmente para crear **páginas web dinámicas** en cliente.
- A día de hoy, es el lenguaje que admite todo navegador web, que interpreta el código cargado por nuestras páginas en el equipo del cliente, del usuario que accede al sitio.
- Una página web dinámica es aquella que produce o modifica su salida en función de la ejecución de un cierto código en el momento de cargar la página, responder a eventos de teclado o ratón, etc.
- A pesar de su nombre, JavaScript no guarda ninguna relación directa con el lenguaje de programación Java.

# Introducción

En origen, el lenguaje JavaScript:

- Era una de las múltiples maneras que surgieron para extender las capacidades del lenguaje HTML.
- No era un lenguaje de programación propiamente dicho como C, Scheme, Python, etc., sino un lenguaje de *script* u orientado a documento, como pueden ser los lenguajes de macros que tienen muchos procesadores de texto y hojas de cálculo.
- Era un **lenguaje interpretado** que se incrustaba en una página web HTML. Un lenguaje interpretado significa que las instrucciones las analiza y procesa el navegador en el momento que deben ser ejecutadas. Lo coordina ECMA.

Hoy día JavaScript (ECMAScript) es mucho más que eso, se usa en cliente y servidor y en todo tipo de aplicaciones. No obstante, en este tema nos centraremos en las características del lenguaje para su uso en cliente, incrustado en el código que se ejecuta dentro del navegador.

# Glosario

- **Script:** cada uno de los programas, aplicaciones o trozos de código creados con el lenguaje de programación JavaScript. A veces se traduce al español directamente como *guión*, aunque script es una palabra comúnmente aceptada.
- **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 y que por tanto no pueden ser utilizadas libremente. Por ejemplo, `if`, `else`, `while`, ...

Nota: el lenguaje va evolucionando en versiones, siendo la actualización más importante ECMAScript 6 (2015).

# Glosario

## Tecnologías relacionadas

- **AJAX**: llamadas asíncronas a servidor sin recarga total de la página.
- **jQuery**: biblioteca de funciones para facilitar funciones AJAX, de acceso a DOM, de eventos, etc.
- **NodeJS**: JS en el lado del servidor.
- **JSON**: notación de objetos de JS, formato de intercambio de ficheros y de interacción cliente-servidor.

Para más información, ver este [curso completo](#) (curso detallado en YouTube, muy bien documentado y desarrollado).

# Frameworks JS

Un framework JS proporciona un conjunto de funcionalidades, una filosofía y una manera de construir aplicaciones, empleando en su núcleo JavaScript (o TypeScript, versión fuertemente tipada que se puede traducir a JS). Los principales (aunque algunas son bibliotecas, no frameworks) son:

- React
- Angular
- Vue
- Svelte

Trataremos de ver Vue, por ser el que tiene una curva de aprendizaje más suave entre los tres más empleados a día de hoy.

# ¿Cómo trabajar con JS?

Para escribir código JavaScript, podemos emplear:

- Un editor de texto, como VisualStudioCode, Atom, SublimeText, Notepad++, Emacs o cualquier otro.
- La propia consola de JavaScript del navegador.
- Un terminal, siempre que hayamos instalado [NodeJS](#).

Para ejecutarlo, disponemos de:

- El navegador, mediante la carga de páginas que contengan JavaScript o mediante la consola.
- Un terminal con NodeJS, como se indicó anteriormente.

También podemos trabajar 100 % online con una plataforma como [JSFiddle](#), [CodePen](#), [PlayCode](#) o similar.



# Introducción

## Un primer ejemplo

```
<html>
  <head>
    <meta charset="UTF-8">
  </head>
  <body>
    <script type="text/javascript">
      document.write('Hola Mundo');
    </script>
  </body>
</html>
```

# Introducción

- El programa JavaScript puede ir directamente en el documento html, en la etiqueta `<script>` , opcionalmente con el atributo

`type="text/javascript"` (hoy día en desuso):

```
<script>
  /* Aquí vendría el código JavaScript */
</script>
```

o en fichero aparte, con `<script src="miscript.js"></script>` .

- Para imprimir caracteres como salida en la página debemos llamar al comando `write` del objeto `document` . La cadena a mostrar debe ir entre comillas y entre paréntesis. Este texto aparecerá en la página HTML resultante. Así, el ejemplo anterior mostrará el texto `'Hola Mundo'` en el navegador.
- Cada instrucción finaliza con punto y coma (recomendable, no obligatorio), y los bloques de instrucciones van entre llaves { y }.
- JavaScript es *case sensitive*, o sensible a mayúsculas y minúsculas ( `var1` es distinto de `Var1` ).

# Introducción

- A la función `write` debemos pasarle como cadena no solo texto plano sino también las etiquetas HTML que necesitemos, de forma que si por ejemplo queremos pasar el siguiente dato a una nueva línea debemos insertar `<br>` :  
`document.write('<br>')` , quedando en el HTML creado como consecuencia de la ejecución del script.
- Esta idea es **fundamental**: En lugar de escribir el código HTML, escribimos sentencias JavaScript que lo escriban en función de su ejecución, pudiendo cambiar el resultado dependiendo de las instrucciones ejecutadas y los parámetros empleados.

```
<html>
  <head>
</head>
  <body>
    <script type="text/javascript">
      document.write('Esto va arriba');
      document.write('<br>');
      document.write('y esto va debajo');
    </script>
  </body>
</html>
```

- `document.write("texto")` : texto a la salida.
- `window.alert("texto")` : mensaje emergente.
- Podemos usar la propiedad `innerHTML` del objeto devuelto por el método `getElementById(Id)` (un nodo HTML).
- Podemos imprimir a la consola de depuración mediante `console.log("texto")`.

# Mensaje de texto

- El siguiente ejemplo muestra el carácter dinámico de JS.
- Es un primer ejemplo de **evento**: *onclick*.
- Al clicar el botón ejecuta la acción *document.write*.
- Nótese que toda la página se reescribe.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Otra página sencilla</h1>
    <p>Un párrafo</p>
    <button type="button"
      onclick="document.write('Sorpresa')">
      Haz la prueba
    </button>
  </body>
</html>
```

# Mensaje de texto

- El siguiente es un ejemplo de `window.alert` .
- Nótese que en este caso no se reescribe la página, sino que el texto sale en una ventana emergente.
- Puesto que no está asociada a ningún evento, la ventana auxiliar con el mensaje sale al cargar la página.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Una página sencilla</h1>
    <p>Un ejemplo de window.alert</p>
    <script>
      window.alert('Hola, mundo');
    </script>
  </body>
</html>
```

# Mensaje de texto

- También podemos asociar el `window.alert` a un evento.
- Por ejemplo, el evento `onmouseenter` tiene efecto cuando el ratón entra dentro del objeto en la pantalla.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
  </head>
  <body>
    
    <p>Pasa el ratón por la imagen</p>
  </body>
</html>
```



# Mensaje de texto

- La forma usual de definir la acción asociada a un evento es mediante una función (manejador de evento) que definimos.
- Veamos un ejemplo sencillo definiendo una función.
- Como vemos, podemos omitir `window.` ante `alert`.

```
<html>
  <body>
    <p>Pulsa para obtener un mensaje de alerta.</p>
    <button onclick="myFunction()">Pulsa</button>
    <script>
      function myFunction() {
        alert("Hola");
      }
    </script>
  </body>
</html>
```

```
<body>
  <h1>Otra página más</h1>
  <p id="demo">Hola</p>
  <button onclick="myFunction()">Pulsa</button>
  <script>
    function myFunction() {
      document.getElementById("demo").
        innerHTML = 'Sorpresa';
    }
  </script>
</body>
```

# Comentarios

Se emplean para proporcionar explicaciones del código que vamos desarrollando. Hay varios tipos de comentarios aceptados en el lenguaje JavaScript:

- Si ocupan una línea, lo ponemos detrás de las barras dobles

```
// Comentario de una línea .
```

- Si son más largos usamos los símbolos `/* */`.

```
/* Comentario  
que ocupa  
varias líneas */
```

A continuación se aportan contenidos actualizados sobre distintos aspectos de JS, si bien algunos se encuentran más detallados (aunque algo antiguos) en [este documento](#).

# Índice

- 1 Introducción
- 2 Variables y tipos
- 3 Instrucciones y funciones
- 4 DOM y eventos
- 5 Miscelánea

# Variables

- Las variables permiten declarar nombres que referencian a una zona de memoria donde alojar datos mientras se ejecute el código donde la variable se encuentra dentro de alcance.
- En JS deben comenzar por letra o subrayado (\_).
- Tienen tipado dinámico (podemos asignar un valor de un tipo a una variable, y luego asignarle un valor de otro tipo).
- No puede tener el nombre de una palabra clave del lenguaje.
- Existen 3 formas distintas de declarar variables o constantes:

```
var x = 10; // Alcance de programa o función  
let y = "Hola"; // Alcance de bloque  
const e = 2.71828; // No asignable de nuevo
```

# Variables

- Hasta 2015, una variable se definía con la palabra `var`:

```
var dia;
```

- Se pueden **declarar** varias variables en una misma línea:

```
var dia, mes, calle;
```

- Una variable se puede declarar e **inicializar** en un paso:

```
var edad = 20;
```

- O en dos pasos (**declaración** e **inicialización** - asignación/modificación -):

```
var edad;  
edad = 20;
```

- También podemos **modificarla** en un momento posterior:

```
edad = 21;
```

# Introducción

```
<html>
  <head>
</head>
  <body>
    <script type="text/javascript">
      var nombre = 'Juan';
      var edad = 63;
      document.write(nombre);
      document.write('<br>');
      document.write(edad);
    </script>
  </body>
</html>
```

# Variables y constantes en ES6/2015

Como adelantamos, en 2015 hay una evolución significativa de la especificación de JavaScript, conocida como ES6 o ES2015 (ES: ECMAScript, estandarizada por ECMA Internacional). Esta evolución afecta al uso de variables:

- Desaconseja el uso de `var` en la mayoría de situaciones.
- Promueve el uso de `let` para variables.
- Define además `const` para constantes.
- Puede ver una descripción detallada en [este enlace](#), o en [este video](#).



# Interacción básica de usuario

## Introducir datos

- Para la entrada de datos por teclado tenemos la función `prompt`. Cada vez que necesitamos introducir un dato con esta función, aparece una ventana donde cargamos el valor.
- La sintaxis de la función `prompt` es:  

```
var_objetivo = window.prompt(mensaje,valor_defecto);
```
- Como vemos, `prompt` tiene dos parámetros: uno es el mensaje a mostrar en la ventana y el otro el valor inicial que aparece en la misma.

# Interacción básica de usuario

## Introducir datos

```
<html>
  <body>
    <script>
      let nombre;
      nombre = window.prompt("Dime tu nombre:", "");
      document.write("Hola ");
      document.write(nombre);
    </script>
  </body>
</html>
```

# Interacción básica de usuario

## Confirmar acciones

- Para confirmar una acción tenemos la función `confirm`.
- La sintaxis de la función `confirm` es:

```
confirmado = window.confirm(mensaje);
```

- Como vemos, `confirm` tiene como único parámetro el mensaje emergente con el que la ventana pedirá confirmación al usuario.
- La función devolverá un valor lógico que indique si queremos o no continuar con la operación que corresponda.

# Interacción básica de usuario

## Confirmar acciones

```
<html>
  <body>
    <script>
      let confirma_accion =
        window.confirm("¿Seguro que quiere saludo?");
      if (confirma_accion) {
        console.log("¡Hola!");
      } else {
        console.log("No se preocupe, no le saludo.");
      }
    </script>
  </body>
</html>
```

# Tipos de datos y objetos predefinidos

Los tipos básicos, primitivos en JavaScript son:

- Número/ `Number` ( `let num = 7.2;` )
- Cadena/ `String`  
( `let txt = "Hola", txt2 = 'Adiós';` ) o  
`let txt3 = `Te saludo con un ${txt}`;`
- Lógico/ `Boolean` ( `let resuelto = false;` )
- Tipos/valores especiales: `null` y `undefined`.
- El tipo `Symbol`, que existe desde ECMAScript 6.
- El nuevo tipo `BigInt`, para enteros que se salgan del rango cubierto por el tipo `Number`.

Además de ellos, tenemos el tipo `Object`, que se crean con el constructor `Object()` o mediante la [notación literal](#), como JSON.

# Tipos de datos y objetos predefinidos

En los siguientes enlaces puede encontrar más información detallada sobre los tipos de datos.

- [Video en Youtube \(DorianDesings\)](#) sobre tipos de datos.
- [Tipos de datos y conversiones de tipos.](#)
- [Tipos de datos \(w3schools\)](#)

# Operadores

Una lista de enlaces relacionados:

- [Video \(de DorianDesings\)](#) sobre operadores básicos sobre distintos tipos de datos.
- <https://uniwebsidad.com/libros/javascript/capitulo-3/operadores>
- <https://es.javascript.info/operators>
- <https://es.javascript.info/comparison>
- <https://es.javascript.info/logical-operators>
- <https://es.javascript.info/nullish-coalescing-operator>

# Funciones sobre tipos de datos

Algunas referencias generales:

- [Objetos predefinidos](#)
- [Funciones y propiedades básicas](#)
- [Tipos de datos \(en detalle, javascript.info\)](#)
- [La palabra clave \*\*this\*\*](#)

Algunas referencias sobre funciones asociadas a tipos específicos:

- [Números \(video sobre objeto Math \(DorianDesings\)\)](#)
- [Cadenas \(video asociado \(DorianDesings\)\)](#)
- [Fechas](#)
- [Arrays y sus métodos](#)
- [Otras estructuras de datos: Conjuntos y Maps \(tablas hash\), y sus funciones de acceso asociadas, Iterables.](#)



# Arrays

Una estructura crucial a la hora de representar aspectos importantes en nuestros sistemas es el array, que permite almacenar una serie de datos en forma de lista ordenada. Para profundizar en los arrays vamos a consultar algunas fuentes importantes:

- [El tipo array \(lenguajejs.com\)](https://lenguajejs.com)
- [Arrays \(javascript.info\)](https://javascript.info) y [sus métodos](#)
- [JS Arrays \(w3schools\)](https://www.w3schools.com/js/js_arrays.asp).

# Arrays

## Aspectos fundamentales

Resumen de los aspectos fundamentales a dominar sobre los arrays

- Creación, mediante `new Array()` o mediante `[ ]`.
- Propiedad longitud `length` y acceso a elemento i-ésimo con operador `a[i]`
- Adición con `push/unshift` y eliminación con `pop/shift`.
- Conversión desde otros elementos con `Array.from(estructura)`.

# Objetos

Podemos [crear objetos básicos](#), con múltiples elementos dentro de los mismos, mediante el operador `{}` (similar a [JSON](#)).

```
const persona = {  
  nombre: "Pepito Pérez",  
  edad: 21,  
  muestraInfo: function () {  
    return `Me llamo ${this.nombre} y tengo ${this.edad} años.`;  
  }  
};
```

Podemos ver más ejemplos y explicaciones detalladas en [este video](#). No hay que confundir este mecanismo básico con la [Programación Orientada a Objetos](#), que se basa en la definición de [clases](#), tiene una importante carga teórica asociada y no vamos a cubrir de momento.

# Índice

- 1 Introducción
- 2 Variables y tipos
- 3 Instrucciones y funciones
- 4 DOM y eventos
- 5 Miscelánea

# Estructuras secuenciales

- Cuando en un programa sólo participan operaciones, entradas y salidas se la denomina estructura secuencial.
- El problema anterior, donde se introduce el nombre de una persona se trata de una estructura secuencial.
- Por ejemplo, podemos pedir dos números por teclado e imprimir su suma.
- Sólo debemos tener en cuenta que si queremos que el operador `+` sume los contenidos de los valores numéricos ingresados por teclado, debemos llamar a la función `parseInt` y pasar como parámetro las variables `valor1` y `valor2` sucesivamente.

# Introducción

```
<body>
  <script type="text/javascript">
    let valor1;
    let valor2;
    valor1 = prompt("Dame el primer número:", "");
    valor2 = prompt("Dame el segundo número", "");
    let suma = parseInt(valor1) + parseInt(valor2);
    document.write("La suma es ");
    document.write(suma);
  </script>
</body>
```

# Introducción

- Cuando se presenta una elección, necesitamos una estructura condicional.
- En una estructura condicional simple, si se verifica la condición se realiza una acción y si no, no ocurre nada.
- Usaremos la instrucción `if` en el lenguaje JavaScript. La condición debe ir entre paréntesis. Si la condición se verifica verdadera se ejecuta todas las instrucciones que se encuentran encerradas entre las llaves de apertura y cerrado seguidas al `if`.
- Podemos utilizar alguno de los siguientes operadores relacionales: `>`, `>=`, `<`, `<=`, `!=` (distinto), `==` (igual), `===` (igual en valor y tipo).
- En la condición del `if` deben intervenir una variable un operador relacional y otra variable o valor fijo.
- Ojo al uso del operador `+` para cadenas de caracteres.

# Introducción

```
<script type="text/javascript">
  let nombre, nota1, nota2, pro;
  nombre = prompt('Nombre:', 'Escribe aquí');
  nota1 = prompt('Primera nota:', 'Escribe aquí');
  nota2 = prompt('Segunda nota:', 'Escribe aquí');
  nota1 = parseInt(nota1);
  nota2 = parseInt(nota2);
  pro = (nota1 + nota2) / 2;
  if (pro >= 5) {
    document.write(nombre + ' aprueba con ' + pro);
  }
</script>
```



# Condicional compuesta

- En general, queremos que si se verifica la condición se realice una acción y que se realice otra si la condición no se satisface.
- La estructura condicional compuesta se construye como:

```
if (cond) {  
    //Instruccion(es) si se cumple cond  
} else {  
    //Instruccion(es) en otro caso  
}
```

- O si hay más casos:

```
if (cond1) { //Instruccion(es) si se cumple cond  
} else if (cond2){ //Instruccion(es) en otro caso  
} else {  
}
```

# Introducción

```
<script type="text/javascript">
  let num1, num2;
  num1 = prompt('Dame un número:', '');
  num2 = prompt('Dame otro número:', '');
  num1 = parseInt(num1);
  num2 = parseInt(num2);
  if (num1 > num2) {
    document.write('El mayor es '+num1);
  } else {
    document.write('El mayor es '+num2);
  }
</script>
```

# Operadores lógicos

- Conjunción (y): &&
- Disyunción (o): ||

```
<script type="text/javascript">
  var num1,num2;
  num1=prompt('Dame un número:', '');
  num2=prompt('Dame otro número:', '');
  num1=parseInt(num1);
  num2=parseInt(num2);
  if ((num1>num2) && (num1> 2 || num2 > 3)) {
    document.write('Sí se cumple');
  } else {
    document.write('No se cumple');
  }
</script>
```

# Estructuras de control: while

```
while (cond) { /* Instrucciones */ }
```

- En primer lugar se verifica la condición; si esta resulta verdadera, se ejecutan las operaciones indicadas entre llaves.
- En caso que la condición sea falsa, la ejecución continúa con la instrucción siguiente al bloque encerrado entre llaves.
- El bloque se repite MIENTRAS la condición sea verdadera.
- Si la condición siempre se cumple, estamos en presencia de un ciclo repetitivo infinito.

# Introducción

```
<script type="text/javascript">
  var x;
  x=1;
  while (x<=100)
  {
    document.write(x);
    document.write('<br>');
    x=x+1;
  }
</script>
```

# Introducción

```
<script type="text/javascript">
  var x=1;
  var suma=0;
  var valor;
  while (x<=5) {
    valor=prompt(`Dame un valor ${x}:`,`Suma 5 valores`);
    valor=parseInt(valor);
    suma=suma+valor;
    x=x+1;
  }
  document.write(`La suma es ${suma}<br>`);
</script>
```

# Estructuras de control: for

- Sintaxis de las estructuras for:

```
for (inic; cond; inc) {  
    /* Conjunto de instrucciones */  
}
```

- Como vemos, esta estructura repetitiva tiene tres argumentos: variable de inicialización, condición y operación de incremento o decremento.

# Ejemplo

- Para escribir los números del 1 al 10, iniciamos la variable `f` como 1.
- A continuación escribimos la condición que debe verificarse `f<=10`. Como la condición se verifica como verdadera se ejecuta el bloque del `for` (en este caso mostramos el contenido de la variable `f` y un espacio en blanco).
- Tras ejecutar el bloque, pasa al tercer argumento del `for` (en este caso, con el operador `++` se incrementa en uno el contenido de la variable `f`).



# Introducción

```
<script type="text/javascript">
  var f;
  for(f=1;f<=10;f++) {
    document.write(f+" ");
  }
</script>
```

# Funciones

- Tienen la siguiente estructura:

```
function nombre_func(arg_1, ..., arg_n) {  
    /* Código de  
    la función */  
}
```

- También pueden ser definidas con expresiones de función:

```
var nombre_func = function (arg_1, ..., arg_n) {  
    /* Código de  
    la función */  
};
```

Puede ver una explicación sobre definición y uso de funciones en:

<https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Funciones>

# Funciones

```
<script type="text/javascript">
  function mostrarComprendidos(x1,x2) {
    for (var inicio=x1; inicio<=x2; inicio++) {
      document.write(inicio+' ');
    }
  }
  var valor1, valor2;
  valor1 = prompt('Dame el valor inferior:', '');
  valor1 = parseInt(valor1);
  valor2 = prompt('Dame el valor superior:', '');
  valor2 = parseInt(valor2);
  mostrarComprendidos(valor1,valor2);
</script>
```

# Definición de funciones

Podemos **crear funciones** mediante:

- Declaración:

```
function alerta () {window.alert('¡Sorpresa!');}
```

- Expresión:

```
var alerta = function () {window.alert('¡Sorpresa!');}
```

- Flecha:

```
var alerta = () => {window.alert('¡Sorpresa!');}
```

*Mientras que las dos primeras instrucciones son parte de JS desde hace décadas, la **notación de flecha** pertenece a EcmaScript 6 (2015). Puede encontrar más información en [esta entrada de blog](#). Para más información se recomienda ver [este video](#).*

# Estructuras de control

## Repaso estructuras empleadas

Hemos visto anteriormente en el tema algunas instrucciones selectivas:

```
if (cond) { instrucs } else { instrucs }
```

Además de instrucciones repetitivas:

```
while (cond) { instrucciones }
```

```
for (inic; cond; incr) { instrucciones }
```

Pero vamos a complementarlo con instrucciones adicionales en las siguientes diapositivas.

# Estructuras de control adicionales

## Instrucciones selectivas

### Switch

```
switch(x) {  
  case 'valor1':  
    // instrucs  
    break;  
  case 'valor2':  
    // instrucs  
    break;  
  default:  
    // instrucs  
}
```

### Operador ternario

```
// Dependiendo del valor lógico "debo_sumar", sumamos o restamos a, b  
let resultado = (debo_sumar) ? (a+b) : (a-b);
```

# Estructuras de control adicionales

## Instrucciones repetitivas

### For recorriendo objetos y arrays

```
for (clave in objeto) {  
    console.log(objeto[clave]);  
}  
  
for (pos in mi_array) {  
    console.log(mi_array[pos]);  
}  
  
for (val of mi_array) {  
    console.log(val);  
}
```

# Estructuras de control adicionales

## Otras estructuras

```
do {  
    // instrucciones  
} while (cond)
```

## Palabras clave distinguidas (*para cualquier bucle*)

- **break** : se sale completamente del bucle más interno en el que está incluido.
- **continue** : se sale de la iteración actual del bucle más interno en el que está incluido.



# Funciones de orden superior

Las funciones de orden superior<sup>1</sup> reciben y/o devuelven funciones, a modo de objetos de pleno derecho. Podemos manejar **estas funciones en Javascript** sin problemas. Veamos algunos ejemplos de uso de algunas funciones de orden superior predefinidas:

```
const y = [1,2,3];  
y.map((i) => 2*i); // devuelve [2, 4, 6]  
y.filter((i) => i%2 == 0); // devuelve [2]  
y.reduce((a,b) => a*10+b); // devuelve 123  
y.forEach((j) => console.log(j**3));  
// Imprime 1, luego 8 y luego 27
```

---

<sup>1</sup>[https://eloquentjavascript.net/05\\_higher\\_order.html](https://eloquentjavascript.net/05_higher_order.html)

# Funciones de orden superior

El cometido de estas funciones es el siguiente:

- `a.map(f)` : aplicado sobre un array existente `a` , devuelve otro array del mismo tamaño y tal que, para cada elemento `x` de `a` , aplique la función unaria `f` sobre `x`, obteniendo `f(x)` .
- `a.filter(p)` : aplicado sobre un array `a` , devuelve otro con aquellos elementos `x` contenidos en `a` que cumplan el predicado unario `p` (es decir, `p(x)` devuelva `true` ).
- `a.reduce(f [, def])` : aplicado sobre un array existente `a` , devuelve el valor resultante de realizar un plegado por la izquierda ( `foldl1` o `foldl` en Haskell, según el caso), empleando la función binaria `f` .
- `a.forEach(pr)` : aplicado sobre un array `a` , ejecutará el procedimiento `pr(x)` para cada elemento `x` de `a` .

## Otras funciones de orden superior

Además de las anteriores, disponemos de otras funciones de orden superior en JS:

- `a.some(p)` : devuelve `true` si existe algún elemento `x` de `a` tal que `p(x)` sea cierto, y `false` en cualquier otro caso (equivalente al `any` de Haskell).
- `a.every(p)` : devuelve `true` si todo elemento `x` de `a` cumple que `p(x)` es cierto, y `false` en cualquier otro caso (equivalente al `all` de Haskell).
- `a.sort(f)` : ordena los elementos del array en función del criterio establecido por la función binaria `f`.

Puede encontrar una explicación detallada y algunos ejemplos prácticos en [este video](#).

# Índice

- 1 Introducción
- 2 Variables y tipos
- 3 Instrucciones y funciones
- 4 DOM y eventos
- 5 Miscelánea

# ¿Qué es el DOM?

## Definition (Definición de DOM)

El **DOM** (*Document Object Model*, o Modelo de Objetos del Documento), es una **API** fundamental (definida a nivel de las tecnologías web, y manejada por el navegador) que permite representar e interactuar con un documento HTML o XML.

Específicamente, cuando trabajamos en un navegador, el **DOM** representa aquello que tenemos cargado en la página, de forma que:

- El DOM del documento es como un árbol, enraizado en document, con un único hijo HTML, que tiene a su vez hijos head y body, y así hasta las hojas (etiquetas sin contenido o texto).
- Podremos acceder a la información de cada nodo de ese árbol, así como crear nuevos nodos o modificar los existentes.

Puede encontrar de una descripción más visual en [este video](#).

# Documentación relacionada

Para algunos aspectos básicos sobre el DOM, podemos hacer uso de los siguientes enlaces

- Objetos del documento (DOM):
  - [Tema](#)
  - [Manual online](#)
  - [Video](#)

# Tipos de nodos

Tenemos muchos [tipos de nodos](#), pero los más comunes son:

- 1 - Elemento HTML (proviniente de cualquier etiqueta HTML cargada en el documento.)
- 2 - Atributo (atributo dentro de una etiqueta).
- 3 - Texto (contenido de texto interno a cualquier etiqueta).
- 8 - Comentario HTML.

Puede encontrar más información en [esta web](#). Nótese que, cuando veamos los nodos hijos de un nodo, generalmente los atributos no se consideran hijos de la etiqueta en la que están presentes (como veremos, estarán contenidos en la propiedad `attributes` del elemento, no en `childNodes` ).

# Acceso al DOM

Podemos **acceder a elementos del DOM** mediante:

- Id: `document.getElementById("miId")`
- Name: `document.getElementsByName("miNombre")`
- Clase: `document.getElementsByClassName("miClase")`
- Etiqueta: `document.getElementsByTagName("p")`
- Selector: `document.querySelector("selSimple")` o  
`document.querySelectorAll("selMult")`

Y asignar a variable: `var elto = document.querySelector("sel")`

Ojo: puedo hacerlo sobre `document` o sobre un nodo cualquiera.

Nota: *los 4 primeros llevan décadas, pero `querySelector` y `querySelectorAll` son más recientes, y reciben como parámetro un selector CSS (excluye pseudoelementos - más detalle en [este sitio](#)).*



# Más funcionalidad asociada al DOM

En [este enlace](#) tenemos muy bien estructurada la información concerniente a las herramientas de las que nos dota JavaScript para trabajar con elementos del DOM, y que desglosa acertadamente en los siguientes grupos:

- [Búsqueda de elementos](#) del DOM, como hemos indicado en la diapositiva anterior.
- [Creación de elementos](#) del DOM.
- [Inserción de elementos](#) y actualización de elementos existentes.
- [Manipulación de CSS](#).
- [Navegación por el árbol](#) de elementos del DOM.

# Más funcionalidad asociada al DOM

## Creación de elementos

Las principales funciones para [crear elementos](#) son:

- `document.createElement(etiqueta)` : esta función permite [crear elementos](#) de la etiqueta indicada.
- `document.createComment(texto)` : crea un nodo de comentario con el texto indicado.
- `document.createTextNode(texto)` : crea un nodo de texto.
- `miNodo.cloneNode(enProf)` : crea una copia de un nodo existente (en profundidad si le pasamos un `true`, y copiando solo los elementos al primer nivel, en caso contrario).
- Propiedad `miNodo.isConnected` : indica si mi nodo está en el DOM o no (puede existir pero no estar en DOM).

Ojo: estas funciones crean elementos, pero necesitamos aplicar otras para añadirlos al DOM.

## Atributos

- `hasAttributes()` : indica si el elemento HTML tiene atributos.
- `hasAttribute(attr)` indica si tiene el atributo indicado.
- `getAttributeNames()` : devuelve array con los nombres de atributos.
- `getAttribute(attr)` : obtiene el valor del atributo (NULL si no existe).
- `setAttribute(attr, val)` : asigna un valor al atributo.
- `removeAttribute(attr)` : elimina el atributo.

Además, podemos acceder directamente a determinados atributos, como `id`, `value` o `style`, con el operador punto (`.`)

## Manipular elementos del DOM

manipular elementos del DOM disponemos de propiedades como:

- `innerHTML` : para consultar o modificar el código HTML contenido entre la apertura y el cierre de la etiqueta de un elemento.
- `outerHTML` : similar al anterior, pero incluyendo al propio elemento.
- `innerText` y `textContent` : para asignar o consultar el texto interno a un elemento (ignora etiquetas como `span` que pueda tener aplicado el texto, devuelve solo el propio texto), con pequeñas diferencias de formateo entre ellas.

Como vemos en el enlace anterior existen otras propiedades, pero su uso no está tan extendido o directamente no son aceptadas por todos los navegadores más empleados.

# Más funcionalidad asociada al DOM

## Incorporar elementos al DOM

Para **incorporar elementos al DOM** disponemos de funciones como:

- `existente.appendChild(nuevo)` : Añade el nodo “nuevo” como hijo del nodo “existente” (detrás de los demás hijos que tenga, en su caso), y devuelve el nodo insertado.
- `insertAdjacentElement(pos, nodo)` : inserta el nodo en la posición indicada ( `"beforebegin"` , `"afterbegin"` , `"beforeend"` o `"afterend"` ).
- `insertAdjacentHTML(pos, cod)` : similar al anterior, pero en lugar de un nodo le pasamos una cadena de código HTML.
- `insertAdjacentText(pos, cod)` : similar, pero añadiendo texto plano.
- `insertBefore(nuevo, existente)` : inserta el nodo justo antes del existente.

# Más funcionalidad asociada al DOM

## Eliminar elementos del DOM

Podemos **eliminar elementos del DOM** con funciones como:

- `nodo.remove()` : elimina el nodo.
- `nodo.removeChild(hijo)` : elimina el hijo indicado.
- `nodo.replaceChild(nuevo, antiguo)` : reemplaza el hijo indicado.

# Más funcionalidad asociada al DOM

## Navegar a través de los elementos HTML

A partir de un cierto elemento del DOM, podemos irnos moviendo hacia otros elementos (hijos, padre, etc.). Disponemos para [esta navegación](#) de estas propiedades:

- `nodo.children` : elementos hijos del nodo.
- `nodo.parentElement` : elemento padre del nodo.
- `nodo.firstChild/lastElementChild` : primer (respec. último) elemento hijo.
- `nodo.previousElementSibling/nextElementSibling` : elemento hermano previo (respec. siguiente) del nodo (NULL si no existe).

# Más funcionalidad asociada al DOM

## Navegar a través de los **nodos**

Lo anterior restringe a elementos HTML, pero hay otros tipos de nodos. Podemos **navegar entre nodos** mediante estas propiedades:

- `nodo.childNodes` : nodos hijos del nodo.
- `nodo.parentNode` : nodo padre del nodo.
- `nodo.firstChild/lastChild` : primer (respec. último) nodo hijo.
- `nodo.previousSibling/nextSibling` : nodo hermano previo (respec. siguiente) del nodo (NULL si no existe).



# Más funcionalidad asociada al DOM

## Manipulación de estilos con JS

Además del [acceso a style](#) de un elemento HTML para manipulación de estilo, podemos [alterar las clases aplicadas](#) de forma dinámica mediante las siguientes propiedades:

- `elto.className` : acceso directo para consultar o modificar la clase de un elemento.
- `elto.classList` : objeto que proporciona la lista de clases, junto con gran cantidad de funcionalidades para alterar la misma:
  - `nodo.classList.add(c1, c2, ...)` : añade las clases indicadas al elemento.
  - `.remove(c1, c2, ...)` : elimina las clases.
  - `.toggle(clase)` : alterna entre añadir/eliminar la clase indicada.
  - `.replace(antigua, nueva)` : reemplaza la aplicación de una clase por otra.
  - `.contains(clase)` : indica si el elemento contiene o no la clase dentro de su lista de clases.
  - `.item(n)` : devuelve la *n*-ésima clase aplicada.

# Formularios

Para poder acceder y manipular los elementos de formulario, que serán cruciales para interacción con el usuario y entre cliente y servidor, disponemos de una serie de objetos, propiedades y funciones. Podemos ver una introducción a estos aspectos en los siguientes enlaces:

- [Tema](#)
- [Manual online](#)
- [Video ilustrativo](#)

# Manejadores de eventos

Podemos manejar un evento sobre un determinado elto. mediante:

- Atributo HTML:

```
<button id="elto" type="button" onclick="alerta();">
```

(puede encontrar muchos ejemplos de uso de este mecanismo en [este documento](#)).

- Acceso JS a DOM y:

- Manejador específico: `elto.onclick = alerta;`

- Mecanismo general:

```
elto.addEventListener('click', alerta, false);
```

Podemos eliminar la asociación mediante:

```
elto.removeEventListener('click', alerta, false);
```

*Mientras que la primera opción es la más fácil de ver al comenzar, lo más aconsejable es utilizar el último mecanismo, para desacoplar contenido de dinámica, entre otras cosas. Puede encontrar más información [aquí](#).*

# Índice

- 1 Introducción
- 2 Variables y tipos
- 3 Instrucciones y funciones
- 4 DOM y eventos
- 5 **Miscelánea**

## Otros elementos no esenciales : Spread operator

- Spread operator ( `...a` ): dado un array `a` con 4 elementos, si tenemos una determinada función definida como `function f (x,y,z,t)`, podemos **llamar** a `f(...a)`, de forma que se extraen los elementos de `a` para asignarse a las variables individuales `x,y,z,t`, en lugar de pasarse `a` como array completo.
- Empleado como parámetros **REST** en la **definición** de una función, `function f (...x)`, la función `f` aceptará cualquier número de parámetros, que se podrán recorrer dentro de la función como si se le hubiera pasado un array. Así, podríamos llamarla con `f(4)`, `f(3,4,5)`, `f(1,2,3,4,5,6)`, etc. (Ver [esta entrada de blog](#)).

Para más información y ejemplos, ver [este video](#).

# Otros ejemplos

- Javascript puede cambiar el contenido HTML.
- Usaremos el método `getElementById` para acceder a los nodos elemento de html por su identificador.
- Usaremos su propiedad `innerHTML` para modificar su contenido.
- Así, el siguiente ejemplo usa el método para encontrar el elemento con identificador `id = cambia` y cambiar su contenido actual por `Esto es una sorpresa`.

# Introducción

```
<body>
<h1>¿Qué puede hacer Javascript?</h1>
<p id="cambia">Puede cambiar el texto HTML</p>
<button type="button"
onclick="document.getElementById('cambia').innerHTML='¡SORPRESA!'">
    Haz la prueba
</button>
</body>
```

# Otros ejemplos

- Javascript puede cambiar una imagen por otra
- En el siguiente ejemplo se combina una función con un condicional `if - else`.



# Introducción

```
<body>
<h1>Este es el escudo de mi equipo</h1>

<p>Haz click sobre él.</p>
<script>
  function changeImage() {
    var image = document.getElementById('miescudo');
    if (image.src.match("escudo_1.jpg")) {
      image.src = "escudo_2.png";
    } else {
      image.src = "escudo_1.jpg";
    }
  }
</script>
</body>
```

# Introducción

```
<h1>Este es el escudo de mi equipo</h1>

<p>Haz click sobre él.</p>
<script>
    function changeImage() {
        var image = document.getElementById('miescudo');
        if (image.width == "100") {
            image.width = "300";
        } else {
            image.width = "100";
        }
    }
</script>
```

# Podemos hacer muchas más cosas ...

```
<body>
<p id="id1" title="Hola"
  style="background-color:red">Ejemplo</p>
<button type="button" onclick="funcionA()">
  Quiero cambiar
</button>
<script>
  function funcionA() {
    if (document.getElementById("id1")
        .innerHTML.match('Ejemplo')) {
      document.getElementById("id1").style =
        'background-color:yellow';
      document.getElementById("id1").innerHTML =
        'Ha cambiado';
    }
  }
}
```

# Otro ejemplo

```
<h1>Elige un botón</h1>
<button type="button" onclick="funcionA()">
  Botón A
</button>
<button type="button" onclick="funcionB()">
  Botón B
</button>
<script>
  function funcionA() {
    window.alert('Has pulsado el botón A');
  }
  function funcionB() {
    window.alert('Has pulsado el botón B');
  }
</script>
</body>
```

## ... mucho más

- Podemos usar la palabra restringida `this` para hacer referencia al objeto sobre el que se define la función.
- El evento `onmouseover` tiene efecto cuando pasamos el ratón sobre el objeto.
- El evento `onmouseout` tiene efecto cuando dejamos de pasar el ratón sobre el objeto.

# Un último ejemplo

```
<head>
  <style type="text/css">
    #miescudo {
      position: absolute;
      top: 100px;
      left: 20px;
    }
  </style>
</head>
```

```
<body>
  <h1>Este es el escudo de mi equipo</h1>
  
  <p>Si quieres cambiarlo, haz click sobre él.</p>
  <script>
    function mueve(x) {
      if (x.style.left == "20px") {
        x.style.left = "250px";
      } else {
        x.style.left = "20px";
      }
    }
  </script>
</body>
```

# Para saber más

- <https://www.tutorialesprogramacionya.com/javascriptya/>
- <http://librosweb.es/libro/javascript/>
- <http://www.w3schools.com/js/default.asp>
- ...

*Estos son únicamente algunos enlaces útiles en relación con el material de la presentación, pero hay mucho más material en la sección de **JavaScript** del sitio web de la asignatura.*

# Hay mucho más que ver...

- Tutorial de JS moderno: <https://javascript.info/>
- Novedades de ES6/2015:  
<https://carlosazaustre.es/ecmascript6>
- Novedades posteriores:  
[https://www.groloop.com/post\\_series/javascript-moderno/](https://www.groloop.com/post_series/javascript-moderno/)
- Petición de datos al servidor



# Programación Orientada a objetos - Clases en JS

- Hemos visto en el tema cómo crear objetos básicos en JavaScript, cada uno pudiendo contener una serie de datos y funciones, dados por sus correspondientes claves.
- Estos objetos básicos o genéricos “cuelgan” del tipo de datos general “Object”, no perteneciendo a ningún otro tipo de datos más específico.
- Pero dentro de JavaScript, como en otros lenguajes, ya podemos trabajar con conceptos de la Programación Orientada a Objetos (POO), para lo que es esencial trabajar con clases.
- Una clase viene a definir un tipo de datos, a modo de la plantilla, que rige qué atributos/propiedades podrá almacenar cualquier instancia de la clase (cualquier objeto concreto creado a partir de la misma) y qué métodos/funciones podrán ejecutarse sobre dicha instancia.

# Definición de Clases en JS

Como se detalla en [este artículo](#) y siguientes, la manera de definir una clase en JS es a través `class`, con un constructor que asigne valores a sus atributos, y una serie de posibles métodos:

```
class Animal {  
  constructor(n, especie) {  
    this.nombre = n;  
    this.especie = especie;  
  }  
  hablar() {  
    return `Hola, soy un ${this.especie} me llamo ${this.nombre}`;  
  }  
}
```

En este ejemplo, el constructor recibe un nombre y una especie y los asigna a atributos. Además, dispone de un método `hablar`, que devuelve una cadena de texto accediendo a los atributos.

# Clases y objetos en JS

Es importante dominar la definición e instanciación de clases en JS, cubriendo los aspectos principales: creación con `class`, instanciación con `new`, acceso a elementos dentro de la clase con `this`, herencia de clases con `extend`, definición de miembros privados con `#` y definición de getter/setters con `get` / `set`. En lugar de reinventar la rueda, vamos a remitir a las fuentes que vimos en clase, y donde debemos profundizar:

- [Artículo sobre clases en JavaScript](#)
- [Herencia de clases en JavaScript](#)

Adicionalmente, para reforzar algunos de los conceptos básicos y ver nuevos ejemplos, se recomienda ver [este video](#) y el siguiente que aparece en la lista de Youtube correspondiente.