

# DESARROLLO WEB, APPS MÓVILES Y DESPLIEGUE EN LA NUBE

## Tema 3 JavaScript

Micael Gallego  
@micael\_gallego

- **Introducción**
- Características básicas
- HTML interactivo
- Orientación a objetos avanzada

- **Introducción**
  - JavaScript
  - Librerías JavaScript
  - Frameworks de alto nivel
  - Referencias
- El lenguaje JavaScript
- HTML Interactivo
- Orientación a objetos avanzada

- Las páginas web pueden incorporar interactividad con el lenguaje **JavaScript**
- Con JavaScript se puede modificar la página y ejecutar código cuando se interactúa con ella (a través del modelo de objetos del documento **DOM**)
- También se pueden hacer peticiones al servidor web en segundo plano y actualizar el contenido de la web con los resultados (**AJAX**)

- Es un lenguaje de programación basado en el estándar **ECMAScript** de **ECMA** (otra organización diferente al **W3C**)
- Hay ligeras **diferencias** en la implementación de JS de los navegadores, aunque actualmente todos son bastante **compatibles** entre sí (**en el pasado no fue así**)

<http://www.ecma-international.org/>

- **Versiones: ES5**

- La versión del estándar más utilizada actualmente es EcmaScript es la **5.1 (2011)**, aunque generalmente se la conoce como **ES5**
- Prácticamente todos los **navegadores soportan la mayoría de las características** definidas en el estándar 5.1

<http://kangax.github.io/compat-table/es5/>

- **Versiones: ES6 (ES2015)**

- En Junio 2015 finalizó el desarrollo de **ES6** con una evolución importante del lenguaje
- A última hora decidieron llamarle oficialmente **ES2015**
- Ningún navegador soporta esta versión completamente, aunque sí algunas características
- Existen **convertidores de código JavaScript escrito en ES6 a ES5**

<http://kangax.github.io/compat-table/es6/>

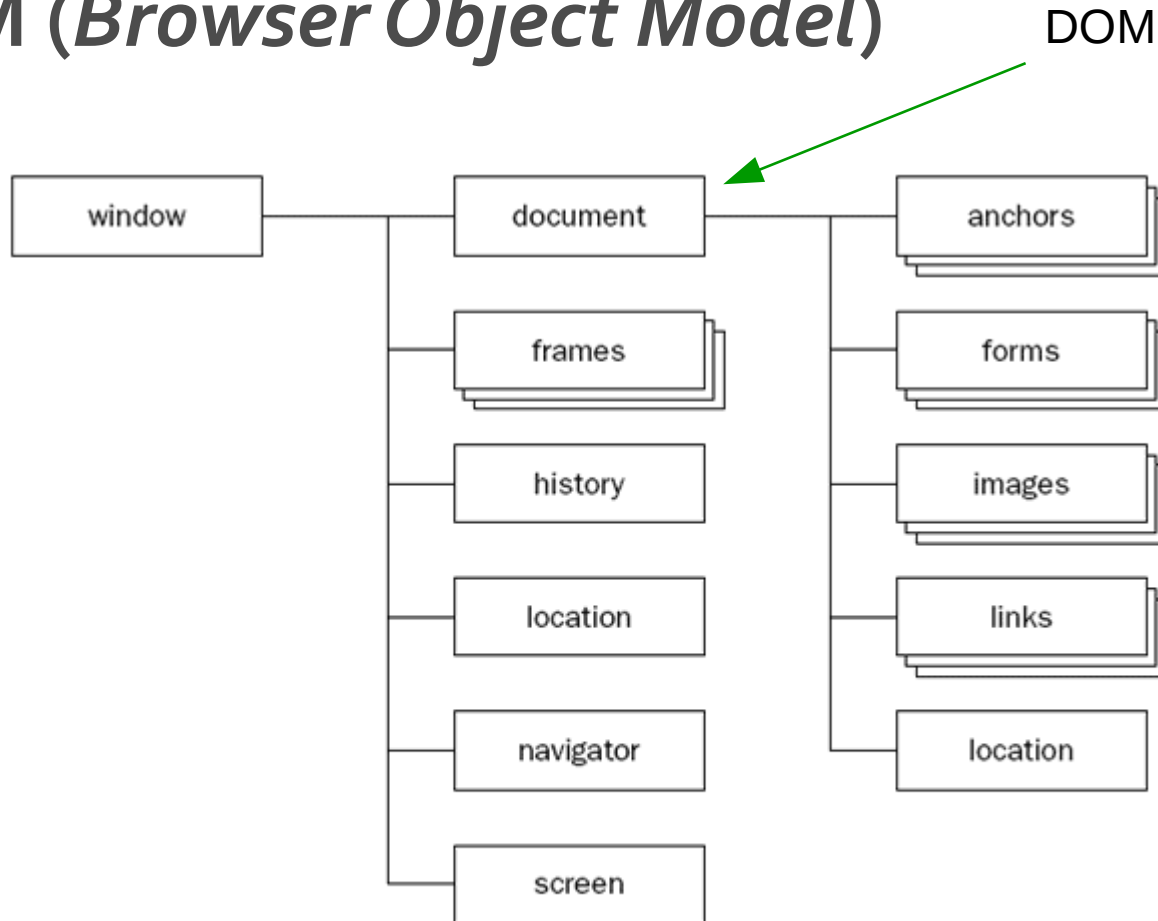
- **JavaScript no es Java**
  - Aunque algunos elementos de la sintaxis recuerden a **Java**, son lenguajes **completamente diferentes**
  - El **nombre JavaScript** se eligió al publicar el lenguaje en una época en la que **Java estaba en auge** y fue principalmente por marketing (inicialmente se llamó **LiveScript**)



- **Características principales de JavaScript**
  - **Scripting:** No necesita compilador. Inicialmente era un lenguaje interpretado, pero actualmente se ejecuta con **máquinas virtuales** en los navegadores (**mayor velocidad de ejecución y eficiencia de memoria**)
  - **Tipado dinámico:** habitual en los lenguajes de script
  - **Funcional:** Las funciones son elementos de primer
  - **Orientado a objetos:** Basado en prototipos, no en clases como Java, C++, Ruby...

- **DOM (*Document Object Model*)**
  - Librería (API) para manipular el documento HTML cargado en el navegador
  - Permite la gestión de eventos, insertar y eliminar elementos, etc.
- **BOM (*Browser Object Model*)**
  - Acceso a otros elementos del browser: historial, peticiones de red AJAX, etc...
  - El BOM incluye al DOM como uno de sus elementos

- BOM (*Browser Object Model*)



# INTRODUCCIÓN

# Librerías JavaScript

- Existen multitud de **bibliotecas** (APIs) JavaScript para el desarrollo de aplicaciones

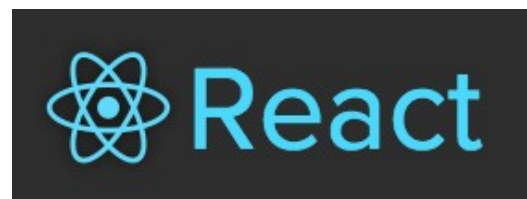


- Algunas de las más usadas son:

UNDERSCORE.JS

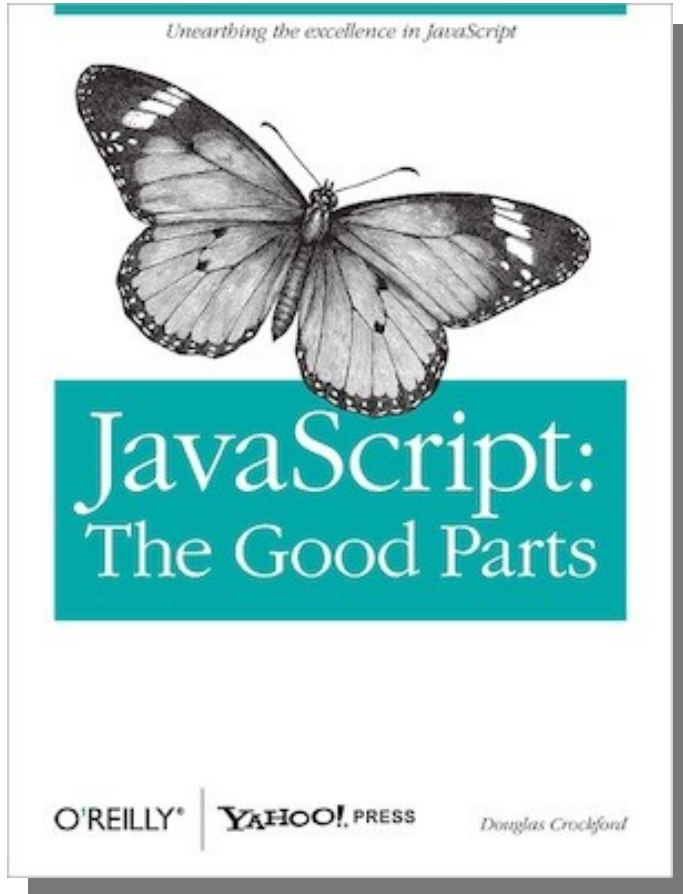
- **jQuery**: es un recubrimiento de la API DOM que aporta facilidad de uso, potencia y compatibilidad entre navegadores. Se usa para gestionar el interfaz (la página) y para peticiones ajax.
- **underscore.js**: Librería para trabajar con estructuras de datos con un enfoque funcional. También permite gestionar plantillas (*templates*) para generar HTML partiendo de datos

- Además de librerías, también existen **frameworks del alto nivel** que estructuran una aplicación de forma completa. Especialmente en **aplicaciones SPA**

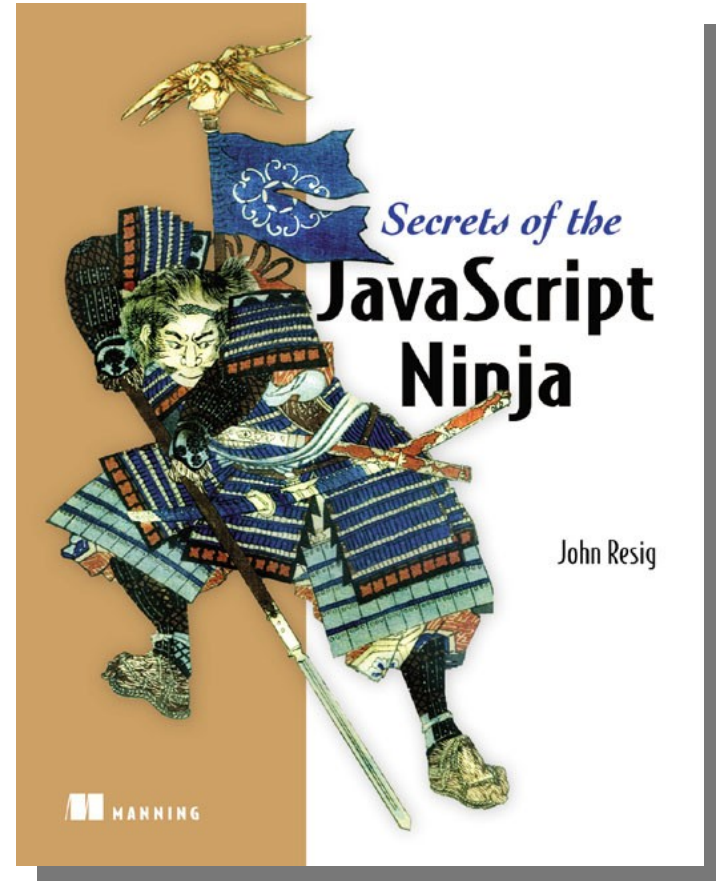


# INTRODUCCIÓN

## Referencias



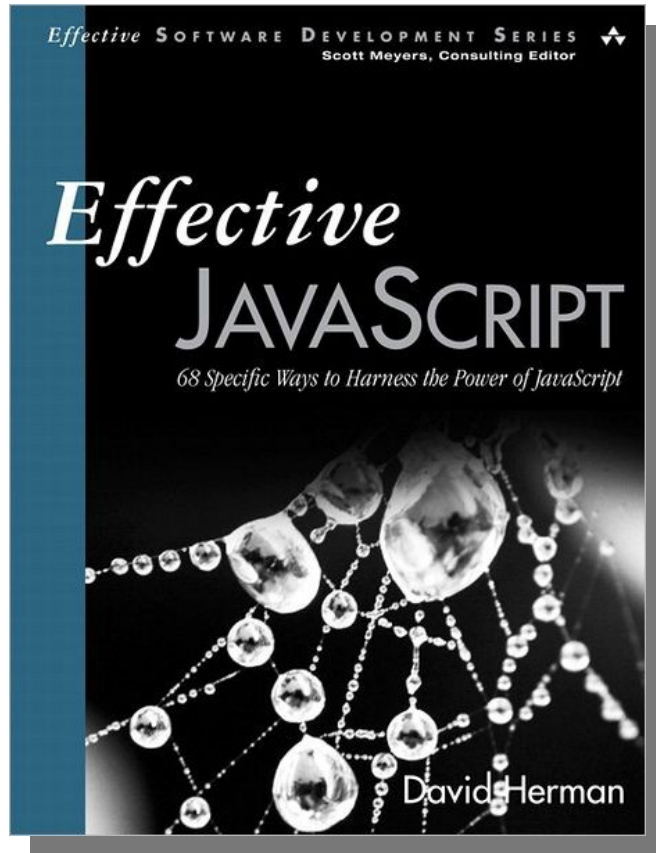
(2008)



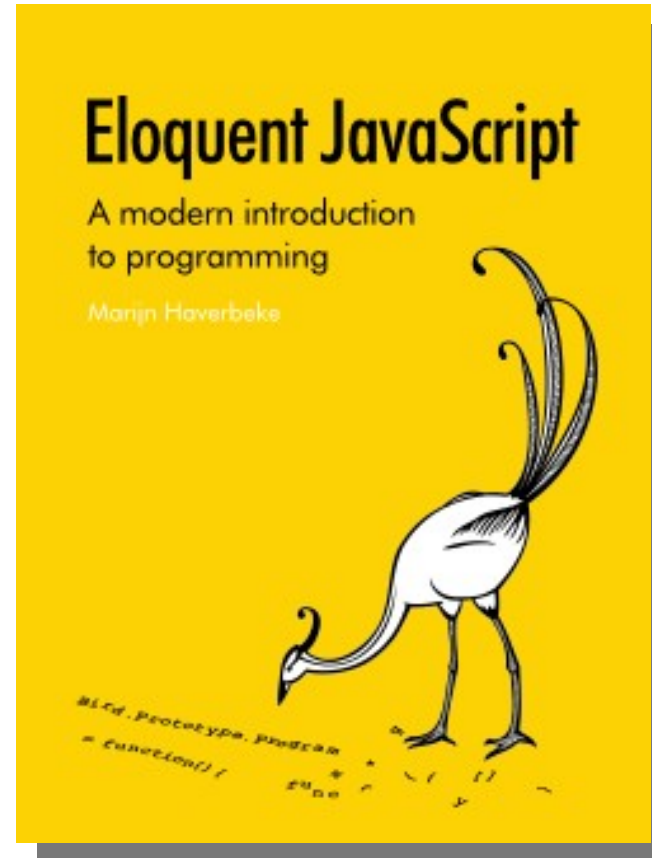
(2012)

# INTRODUCCIÓN

## Referencias



(2012)



(2011) Descarga gratuita

- **Tutoriales interactivos on-line**

- Codecademy.com
- Codeschool.com

- **Blogs**

- Introducción a JavaScript:
  - <http://michaelgallego.github.io/blog/Viaje-por-las-tecnologias-de-front-end2/>
- Comparativa de POO en Java y JS:
  - <http://michaelgallego.github.io/blog/Orientacion-a-Objetos-en-JavaScript-comparado-con-Java/>
- JavaScript no es orientado a objetos:
  - <http://michaelgallego.github.io/blog/JavaScript-no-es-orientado-a-objetos/>



- Introducción
- **El lenguaje JavaScript**
  - Características del lenguaje
  - Integración con HTML
  - Sintaxis básica
  - Arrays
  - Sentencias de control de flujo
  - Funciones
  - Excepciones
  - Orientación a objetos básica
- HTML Interactivo
- Orientación a objetos avanzada

- Vamos a estudiar las principales **características de JavaScript ES5**
- No estudiamos ES6 porque todavía no está muy soportado por los navegadores
- Como **Java** es un lenguaje muy conocido, JavaScript se presentará en **comparación** con éste

- **Imperativo y estructurado (como Java)**
  - Se declaran **variables**
  - Se ejecutan las sentencias **en orden**
  - Dispone de **sentencias de control** de flujo de ejecución (if, while, for...)
  - La **sintaxis** imperativa/estructurada es muy parecida a Java y C

- **Lenguaje de script (Java es compilado)**
  - No existe compilador
  - El navegador carga el código, lo analiza y lo ejecuta
  - El navegador indica tanto **errores de sintaxis** como errores de **ejecución**
- **Tipado dinámico (Java tiene tipado estático)**
  - Al declarar una variable no se indica su tipo
  - A lo largo de la ejecución del programa una misma variable puede tener **valores de diferentes tipos**

- **Orientado a objetos**

- Todos los **valores son objetos** (no como en Java, que existen tipos primitivos)
- Existe **recolector de basura** para liberar la memoria de los objetos que no se utilizan (como en Java)
- La orientación a objetos está **basada en prototipos** (en Java está basada en **clases**)
- En tiempo de ejecución se pueden **crear objetos**, cambiar el valor de los **atributos** e invocar a **métodos** (como en Java)
- En tiempo de ejecución se pueden **añadir y borrar atributos y métodos** (en Java no se puede)

## • Funciones

- Aunque sea orientado a objetos, también permite declarar **funciones independientes** (en Java todas las funciones son métodos de clases)
- Las funciones se pueden declarar en cualquier sitio, asignarse a variables y pasarse como parámetro
- Existen **funciones anónimas** (como las **expresiones lambda** de Java)
- En JavaScript se puede implementar código siguiendo el **paradigma funcional**

## • Modo estricto

- Las primeras versiones de JavaScript permitían escribir código que posteriormente se consideró **propenso a errores**
- La versión **ES5** definió un **modo estricto**, en el que ese código no se considera válido y **genera un error**
- Para **activar** el modo estricto basta poner al principio del código la sentencia (**recomendable**)

```
"use strict";
```

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict\\_mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)

- El código **JavaScript** se puede incluir directamente en el documento HTML en etiquetas **<script>**
- Pero es **recomendable** que el código JS esté en en **ficheros aparte**
- El código **será cargado y ejecutado** cuando se **encuentre** en el documento, antes de continuar procesando el HTML

```
<html>
  <body>
    <scrip>alert("Hola") ;</script>
    ...
    <script src="js/script.js"></script>
    ...
  </body>
</html>
```






- **Velocidad de carga de las páginas**
  - Cuando se carga el JavaScript no se pueden cargar otros recursos en paralelo, por ello, se recomienda poner el elemento `<script>` como **último elemento de la página**

```
<html>
  <body>
    ...
    <script src="js/script.js"></script>
  </body>
</html>
```



- **Velocidad de carga de las páginas**
  - Cuando el código JavaScript no es muy grande y la página tiene muchos elementos interactivos que necesitan JavaScript, es recomendable **cargar el JS cuanto antes**
  - Se puede poner en la cabecera de la página

```
<html>
  <head>
    <script src="js/script.js"></script>
  </head>
  <body>
    ...
  </body>
</html>
```



- **Mostar información desde JavaScript**
  - Escribiendo en el documento HTML en la posición del script

```
document.write('Texto');
```

- Escribiendo en la consola JavaScript del navegador

```
console.log('Texto');
```

- **Comentarios (como en Java)**
  - Una línea y multilínea
- **Delimitadores (como en Java)**
  - De bloque {}
  - De sentencia ; (opcionales)
- **Palabras reservadas**

```
abstract boolean break byte case catch char class const continue
debugger default delete do double else enum export extends false
final finally float for function goto if implements import in
instanceof int interface long native new null package private
protected public return short static super switch synchronized
this throw throws transient true try typeof var volatile void
while with
```

- **Tipos de datos “básicos”**

- Son objetos (no existe distinción como en Java entre objetos y tipos primitivos)
- **Number**
  - Números enteros y reales de cualquier precisión
- **String**
  - Cadenas de caracteres
  - Comillas simples o dobles
  - Inmutable (como en Java)
  - Operador + (como en Java)
  - Como no hay tipo caracter, se usa un string de tamaño 1
- **Boolean**
  - true o false (como en Java)

- **Variables**

- Es un lenguaje dinámico (con tipado **dinámico**)
- Las variables se tienen que declarar pero **no se indica el tipo**

```
//La variable edad tendrá un valor de 34  
var edad = 34;  
var encontrado = false;
```

- Las declaraciones se suelen hacer al **principio de la función**
- La variable tiene como **ámbito la función, no el bloque**. Al finalizar el bloque, la variable sigue presente (En Java es por bloque)

- **Variables**

- Si las variables **no se inicializan** tienen el valor **undefined** (en vez de cero o null como en Java)

```
var hola;  
document.write("Hola: " + hola);
```



Hola: undefined

- Si las variables **no se declaran (por una equivocación)**
  - ▢ Si se intenta **leer su valor** salta un **error** (que finaliza la ejecución)
  - ▢ Si se **asigna un valor**, se crea un nuevo atributo en el objeto global (**no dan error las siguientes lecturas**)

- **Operadores en expresiones**

- **Similares a Java**

- ▯ Aritméticos: + - \* / % (a división es siempre real)
    - ▯ Comparación números: < > <= >=
    - ▯ Lógicos: && || !
    - ▯ Comparativo: ?: (Elvis operator)
    - ▯ Modificación: ++ --
    - ▯ Asignación: = += -= \*= /= %=

- **Diferentes a Java. Comparación**

- ▯ Igual: ===
    - ▯ Distinto: !==
    - ▯ En strings se comporta como el equals(...) en Java
    - ▯ En arrays se comporta como == en Java



- Los arrays de JavaScript son parecidos a los de Java
  - El acceso para lectura o escritura es con `[]`
  - Tienen la propiedad **length**
  - Empiezan por **cero**
  - La **asignación** de variables **no copia** el array, las variables apuntan al **mismo objeto**
  - El operador `===` compara si son el **mismo objeto** (no que tengan el contenido igual)

- Los arrays de JavaScript son parecidos a los de Java
  - Los arrays de varias dimensiones son arrays de arrays. Hay que crear de forma explícita los niveles.
  - Se recorren de la misma forma

```
var numbers = [3, 4, 4, 2];  
  
for (var i = 0; i < numbers.length; i++)  
{  
    document.write(numbers[i] + ',');  
}
```



3,4,4,2,

- **Pero se diferencian en algunos detalles**

- Los literales son con `[]` en vez de `{}` y sin `new`

```
var empty = [];  
var numbers = ['zero', 'one', 'two', 'three']
```

- Para crear un array con varias posiciones se usa

```
var array = new Array(3);
```

- Los arrays pueden mezclar valores de varios tipos (como si fuera un array de `Object` en Java)

- Errores de acceso

- La lectura de un elemento fuera de los límites devuelve **undefined**
  - ▮ En Java sería un `ArrayIndexOutOfBoundsException`
- La escritura de un elemento fuera de los límites del array se permite, haciendo más grande el array y rellenando con valores **undefined**

- **Errores de acceso**

```
var numbers = [3, 4, 4, 2];  
  
document.write(numbers[5] + '<br>');  
  
numbers[7] = 4;  
  
document.write(numbers[7] + '<br>');  
  
for (var i = 0; i < numbers.length; i++){  
    document.write(numbers[i] + ",");  
}
```



```
undefined  
4  
3,4,4,2,undefined,undefined,undefined,4,
```

- Errores de acceso

- El intento de acceso a un **array undefined** da un error "TypeError: Cannot read property '7' of undefined"
  - ▮ En Java sería un `NullPointerException`

- **Modificación del array (como ArrayList de Java)**
  - Se pueden establecer elementos en posiciones no existentes y el array crece dinámicamente.
  - El método **push** añade un elemento al final (como el método **add** en Java)
  - La propiedad **length** se puede cambiar para reducir el tamaño del array

- **Modificación del array (como ArrayList)**
  - El **operador delete** borra un elemento (pero deja el hueco con valor **undefined**)

```
delete numbers[2];
```

- Para borrar y no dejar el hueco se usa el método **splice** indicando el índice desde el que hay que borrar y el número de elementos.

```
numbers.splice(2, 1);
```



- **Bloques de sentencias**

- Con llaves { } (como en Java)
- Las variables no desaparecen al terminar en bloque (en Java si)

- **Sentencia if**

- Sintaxis como en Java
- No es obligatorio que la expresión devuelva un boolean
- Se considera falso: false, null, undefined, "" (cadena vacía), o, NaN

- **Sentencias switch, while, do while**
  - Sintaxis y semántica como en Java
- **Sentencia for**
  - Sintaxis y semántica como en Java
  - Existe el operador **break**
  - No existe el operador **continue**
- **Sentencia return**
  - Como en Java

- **JavaScript** es un lenguaje **funcional** en el sentido de que las **funciones** son **ciudadanos de primera clase**
- Se pueden declarar con nombre

```
function func(param) {  
    console.log(param);  
}  
  
func(4); // Imprime 4
```

- Se pueden declarar **sin nombre (anónimas)** y asignarse a una **variable** o usarse como **parámetro**

```
var func = function(param) {  
    console.log(param) ;  
}  
  
func(4) ; // Imprime 4
```

- En **JavaScript** las funciones son **objetos**
- Por eso se pueden guardar en **variables**, pasar como **parámetro**
- También tienen **métodos** y **atributos** (como cualquier objeto)
- En Java se pueden las funciones se implementan con **expresiones lambda** o con **clases anónimas**

- **Invocación de una función**
  - Nombre seguido de parámetros:

```
func (3) ;
```

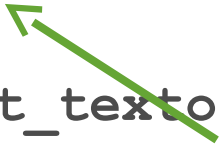
- Si se pasan menos parámetros de los que están en la cabecera, los que faltan toman el valor **undefined** (**No hay error**)
- Si se pasan más parámetros de los que están en la cabecera, los que sobran se **ignoran** (**No hay error**)

- **Información accesible desde la función**
  - Parámetros y variables declaradas en la función

```
var func = function (param){  
    var numero = 0;  
    console.log(param+" numero:"+numero);  
}  
  
func(4); // Imprime '4 numero:0'
```

- Información accesible desde la función
  - Variables accesibles en el punto en que se declara la función

```
var texto = "Hola";  
  
var print_texto = function () {  
    console.log(texto);  
}  
  
print_texto(); // Imprime 'Hola'
```





- **Información accesible desde la función**
  - Cuando se referencia a una variable no sólo se accede a su valor, se accede a la **propia variable en sí**
  - Si se **cambia el valor** de la variable, la **función podrá leer** el nuevo valor
  - En **Java**, si una expresión lambda o una clase anónima accede a una variable de fuera de su declaración, esta variable **no puede cambiar de valor** (daría error de compilación)

- Información accesible desde la función
  - El conjunto de variables a las que tiene acceso la función se llama **cerradura o cierre** (*closure*)

```
var texto = "Hola";  
var print_texto = function () {  
    console.log(texto);  
}  
print_texto(); // Imprime 'Hola'  
texto = "Adios";  
print_texto(); // Imprime 'Adios'
```

- **Funciones dentro de funciones**
  - Se pueden declarar funciones en el cuerpo de otras funciones

```
var add_onclick_handler = function(node)
{
    node.onclick = function (e) {
        alert("Alerta");
    };
};
```

- **Uso de funciones para aislar código**

- Todas las variables declaradas en un script son accesibles en el siguiente script
- Para evitar que haya interferencias no deseadas, el código de un script se suele **encapsular dentro de una función** que se llama inmediatamente (***Immediately-Invoked Function Expression (IIFE)***)

```
(function() {  
  
    var variableTemporal = "a";  
    //Otro código...  
  
})();
```

- Funcionan igual que en **Java**
- Existe un bloque con **try catch finally**
- El operador **throw** eleva la excepción
- A diferencia de Java, se puede lanzar cualquier objeto como excepción, aunque lo recomendable es elevar un objeto con propiedades **name** y **message**

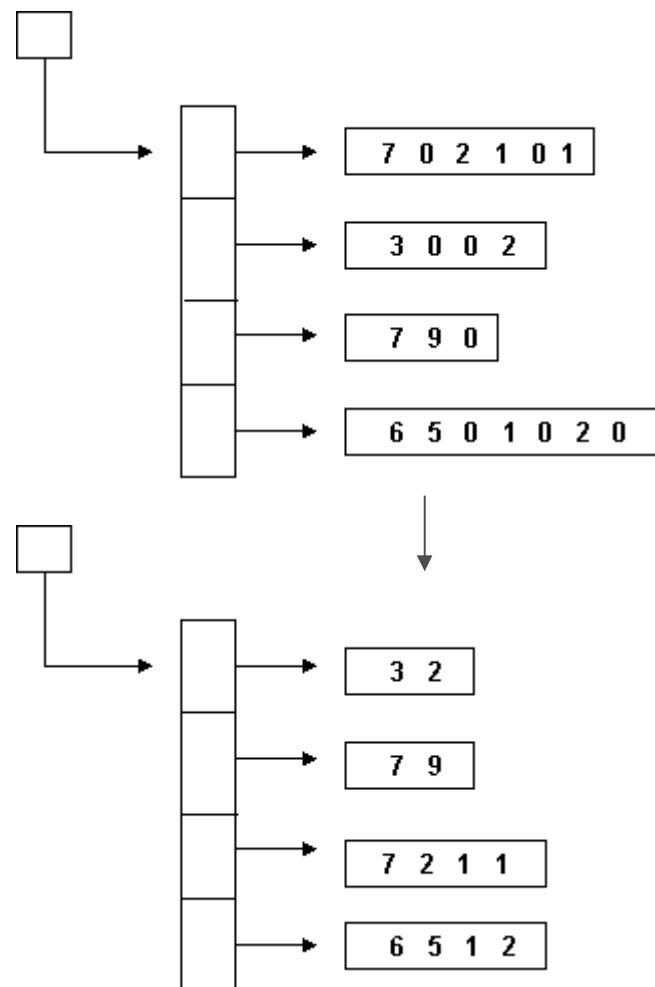
```
try {  
    var error = ...;  
    if (error) {  
        throw "An error";  
    }  
    return true;  
  
} catch (e) {  
  
    alert (e);  
    return false;  
  
} finally {  
    //do cleanup, etc here  
}
```

- Crear una función que **reciba un array como parámetro** y devuelva un array de dos elementos
- El primer elemento apuntará a un **array con los números pares** del array que se pasa como parámetro
- El segundo elemento apuntará a un **array con los números impares** del array que se pasa como parámetro
- Para probar la función se implementarán varias llamadas con **diferentes arrays** y el resultado se mostrará en la **consola del navegador**

# EL LENGUAJE JAVASCRIPT

## Ejercicio 2

- Crear una función que **reciba un array bidimensional, le quite los ceros y ordene las filas de menor a mayor longitud**
- Para probar la función se implementarán varias llamadas con **diferentes arrays** y el resultado se mostrará en la **consola del navegador**





- Para ordenar los arrays por tamaño se puede usar el **algoritmo de la burbuja** (*bubble sort*)
- El algoritmo consiste en comparar cada elemento del array con el siguiente, intercambiándolos de posición si están en el orden equivocado
- El array se recorre hasta que todos los elementos estén ordenados (un elemento siempre es menor que el siguiente)

- Hasta ahora hemos visto que **JavaScript** es un lenguaje bastante familiar en cuanto a **sintaxis** y **características**:
  - Imperativo y estructurado
  - Tipado dinámico
  - Tres tipos básico: Number, Boolean y String
  - Arrays dinámicos
  - Recolector de basura
  - Funciones

- La orientación a objetos en **JavaScript** a primera vista parece **similar** a la de **Java**, pero en esencia es **muy diferente**
- La gran **mayoría de los lenguajes** de programación implementan la Orientación a Objetos con **clases** (Java, C#, C++, Python, Ruby...)
- En JavaScript no existen las clases, existen los **prototipos**
- En el apartado de **orientación a objetos avanzada** se estudiará con detalle esta característica

- Creación de objetos

- En cualquier momento se puede **crear un objeto**, **definir sus atributos** y **asignarles valor**

```
var empleado = {  
  nombre: "Pepe",  
  salario: 700  
}
```

- **Acceso a los atributos de un objeto**

- Notación punto (como en Java)
- **Leer** valor del atributo

```
var salario = empleado.salario;
```

- **Escribir** nuevo valor en un atributo

```
empleado.salario = 800;
```

- **Añadir** un nuevo atributo al objeto en cualquier momento

```
empleado.telefono = "663232539";
```

- Atributos de un objeto

```
var empleado = {  
  nombre: "Pepe",  
  salario: 700  
}  
  
console.log("S:"+empleado.salario);  
  
empleado.salario = 800;  
  
empleado.telefono = "663232539";  
  
console.log("T:"+empleado.telefono);
```

- **Métodos en un objeto**

- Los objetos pueden tener **métodos**
- **Los métodos son en realidad** funciones asignadas a una propiedad

```
var empleado = {  
  nombre: "Pepe",  
  salario: 700,  
  toString: function() {  
    return "N:"+this.nombre+" S:"+this.salario;  
  }  
}
```

Para acceder a los atributos del objeto es necesario usar this (en Java es opcional)

- **Métodos en un objeto**

- Los métodos se **invocan** con la notación punto (como en Java)

```
//Devuelve 'Nombre:Pepe, Salario:700'  
var texto = empleado.toString();
```

- Se pueden **añadir métodos** a un objeto en cualquier momento

```
empleado.getCategoria = function(){  
    return this.salario > 800 ? "Superior":"Normal";  
}
```



- Métodos en un objeto

```
var empleado = {  
  nombre: "Pepe",  
  salario: 700,  
  toString: function(){  
    return "N:"+this.nombre+" S:"+this.salario;  
  }  
}  
  
//Muestra N:Pepe S:700  
console.log(empleado.toString());  
  
empleado.getCategoria = function(){  
  return this.salario > 800 ? "Superior":"Normal";  
}  
  
console.log('C:'+empleado.getCategoria());
```

- Como se puede ver, los **objetos** son muy flexibles porque **no necesitan** un **molde** (una clase)
- Esta forma de trabajar es muy útil cuando **sólo hay un objeto** con una estructura determinada (***singleton***)
- También permite crear **objetos** con **ciertos** atributos y métodos **iguales**, pero con **otros** atributos y métodos **particulares**

- **Objetos, null y undefined**

- En JavaScript también se puede usar **null** como un valor válido para las variables
- Las variables que no están inicializadas tienen el valor **undefined**
- Usar una variable que puede que apunte a un objeto

```
var obj = null;  
if(obj) {  
    obj.doSomething();  
}
```

Devuelve true si  
obj no es **null** ni es  
**undefined**

- **Acceso a las propiedades de un objeto con []**
  - Además de la notación punto (recomendada), también se pueden usar los **corchetes** para acceder a las propiedades de un objeto (atributos y métodos)

```
var obj = {  
    a: "a",  
    metodo: function () { return this.a; }  
};  
  
console.log(obj.a);  
console.log(obj["a"]);  
  
console.log(obj.metodo());  
console.log(obj["metodo"]());
```

- **Objetos como estructuras de datos Mapa**
  - Al poder acceder a las propiedades de un objeto mediante corchetes, se pueden tratar como un **mapa con claves de tipo String**.

```
var config = {}  
config["nombre"] = "Pepe Pérez";  
config["correo"] = "pepe.perez@blabla.com";  
config["idioma"] = "ES";  
  
console.log("Nombre: "+config["nombre"]);  
  
for(key in config){  
    console.log(key+": "+config[key]);  
}
```

Para consultar las propiedades de un objeto se usa **for(in)**

- Hasta ahora hemos visto los **aspectos básicos** del lenguaje de programación **JavaScript**
- Muchos programas se pueden implementar **únicamente con estos elementos**
- Más adelante veremos cómo implementar **programas grandes, estructurando el código** de forma parecida a como se hace en Java con **clases cuyos objetos que colaboran entre sí**

- Introducción
- El lenguaje JavaScript
- **HTML Interactivo**
  - Document Object Model (DOM)
  - jQuery
- Orientación a objetos avanzada

- JavaScript se diseñó para dotar de **interactividad a las páginas HTML** cargadas en el navegador
- Se puede **ejecutar código** JavaScript cuando el usuario interactúa con la página (**click, hover, etc...**)
- Se puede **modificar la página HTML** con JavaScript
  - **Cambio de CSS** de un elemento (ocultar, cambio de color...)
  - **Cambio del contenido:** Texto, imágenes, ...



- El navegador permite acceder al documento y al browser
  - El **Document Object Model (DOM)** es una “librería” para manipular documentos HTML

[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)

- El **Browser Object Model (BOM)** permite manipular otros elementos del navegador

<https://developer.mozilla.org/en-US/docs/Web/API/Window>

<https://developer.mozilla.org/en-US/docs/Web/API/Navigator>

<https://developer.mozilla.org/en-US/docs/Web/API/Location>

- **Eventos**

- Asociar **funciones** a elementos del HTML de forma que serán ejecutadas cuando se interactúe con dichos elementos

```
<html>
  <body>
    <script>
      function alerta() {
        alert('El botón ha sido pulsado');
      }
    </script>
    <button onclick="alerta();" >Botón</button>
  </body>
</html>
```

- **Eventos**

- También se pueden asociar eventos desde el código JavaScript

```
document.addEventListener("DOMContentLoaded", function() {  
    alert('Documento cargado');  
});
```

```
<html>  
  <head>  
    <script src='app.js'></script>  
  </head>  
  <body>  
    ...  
  </body>  
</html>
```

- Acceso a los elementos del documento

- Usando su id

```
var arrImages = [];  
  
arrImages[0] = document.getElementById("image1");  
arrImages[1] = document.getElementById("image2");  
arrImages[2] = document.getElementById("image3");  
  
var objOutput = document.getElementById("output");
```

- También se puede acceder a los elementos en base a
  - ▢ Si tiene asociada una clase (class)
  - ▢ El nombre del elemento
  - ▢ El valor del atributo "name"

- **Acceso o modificación del HTML**

- Modificar el HTML de un elemento

```
var element = document.getElementById("txt");  
element.innerHTML = "<p>Nuevo texto</p>"
```

- Cambiar la clase del elemento

```
document.getElementById("MyElement").className = "MyClass";
```

- Añadir una clase a un elemento

```
document.getElementById("MyElement").className += " MyClass";
```

- **Acceso o modificación del HTML**
  - Cambiar el estilo directamente

```
var element = document.getElementById("img1");  
  
element.style.borderWidth = width + "px";
```

- **El DOM es un estándar**, pero no todos los navegadores implementaban el estándar de forma completa (especialmente IE)
- **BOM no es estándar**, y cada navegador lo implementa de formas ligeramente diferentes
- Siempre hay que consultar la **documentación** y hacer **pruebas con cada navegador**

- **jQuery** es una librería **JavaScript** para lidiar con estos problemas e incompatibilidades
- Además ofrece un interfaz mucho más directo e intuitivo interactuar con el documento y el navegador



<http://jquery.com>



- **Funcionalidades**

- Acceso a los elementos de un documento HTML

```
$('div.content').find('p');
```

- Modificar la apariencia del documento HTML

```
$('ul > li:first').addClass('active');
```

- Alterar el contenido del documento HTML

```
$('#container').append('<a href="more.html">more</a>');
```

- **Funcionalidades**

- Responder a las interacciones del usuario

```
$('button.show-details').click(function() {  
    $('div.details').show();  
});
```

- Animar elementos del documento

```
$('div.details').slideDown();
```

- Descargar información del servidor sin actualizar la página (AJAX)

```
$('div.details').load('more.html #content');
```

- **Características**

- Para seleccionar elementos de la página con **jQuery** usamos la misma sintaxis (selectores) en que en CSS
- Existen funcionalidades adicionales en forma de **plugins**, de forma que la librería se mantiene pequeña
- **Unifica las diferencias** que existen en los navegadores
- Trabaja directamente con **conjuntos de elementos**, de forma que no es necesario procesar cada elemento de forma individual (**no hay que hacer el for**)
- Tiene una **API fluida** (*fluent API*) para reducir el uso de variables locales y poder escribir scripts en una línea

- **Uso de jQuery**

- La librería se puede descargar de su página web para desarrollo local (<http://jquery.com/>)
- Cuando se publica una página web que usa la librería, en vez de servir la librería en el servidor web, es mejor usar una ruta en una **red de distribución de contenido (CDN)**
- Si la librería está cacheada en el navegador del cliente no se descargará de nuevo (reduciendo el tiempo de carga)

<http://code.jquery.com>

<https://developers.google.com/speed/libraries/devguide#jquery>

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="style.css">
    <script src="https://code.jquery.com/jquery-2.2.0.min.js">
    </script>
    <script src="app.js"></script>
  </head>
  <body>
    <h1>Title</h1>
    <p>Text...</p>
  </body>
</html>
```

app.js

```
$(document).ready(function() {  
    $('div.poem-stanza').addClass('highlight');  
});
```

- La función `$` está sobrecargada y admite muchos tipos de parámetros
  - Si se le pasa el **documento**, devuelve un objeto con métodos que permiten asociar funciones que se ejecutarán ante un determinado evento de carga (p.e. **ready**)
  - Si se le pasa un **String**, es un **selector** CSS que devuelve un conjunto con los elementos del documento **seleccionados**

app.js

```
$(document).ready(function() {  
    $('div.poem-stanza').addClass('highlight');  
});
```

- El conjunto de elementos admite métodos que se aplicarán sobre todos los elementos:
  - **addClass(...):** Añade una clase CSS a los elementos
  - **removeClass(...):** Elimina una clase CSS a los elementos

app.js

```
$(document).ready(function() {  
    $('div.poem-stanza').addClass('highlight');  
});
```

- `$(document).ready(...)`
  - Ejecuta la función cuando el documento HTML se ha cargado y antes de que se descarguen las imágenes
  - Permite llamar varias veces a `ready(...)` y asociar varias funciones
  - La función se ejecuta si se llama después de que el documento haya sido cargado



- **Selección con selectores CSS**

- Basados en elementos, clases e ids

```
$(document).ready(function() {  
    $('#selected-plays > li').addClass('horizontal');  
});
```

- Basados en atributos con expresiones regulares

```
$(document).ready(function() {  
    $('img[alt]').addClass('imgstyle');  
    $('a[href^="mailto:"]').addClass('mailto');  
    $('a[href$=".pdf"]').addClass('pdflink');  
    $('a[href^="http"][href*="henry"]').addClass('henrylink');  
});
```

- **Selectores propios de jQuery (no CSS)**
  - Elementos pares (empezando con 0)

```
$(document).ready(function() {  
    $('tr:even').addClass('alt');  
});
```

- Hijos pares (empezando con 1)

```
$(document).ready(function() {  
    $('tr:nth-child(odd)').addClass('alt');  
});
```

<http://api.jquery.com/category/selectors/>

- **Selectores propios de jQuery (no CSS)**
  - Elementos basados en el texto que contienen

```
$(document).ready(function() {  
    $('td:contains(Henry)').addClass('highlight');  
});
```

- Selectores de elementos de formulario

```
$('input[type="radio"]:checked')  
$('input[type="password"], input[type="text"]:disabled')
```

<http://api.jquery.com/category/selectors/>

- Seleccionar elementos con operaciones
  - Filtrar los elementos de un grupo por algún criterio

```
$('tr').filter(':even').addClass('alt');  
$('a').filter(function() {  
    return this.hostname && this.hostname!=location.hostname;  
}).addClass('external');
```

- Navegar por el documento

```
$(document).ready(function() {  
    $('td:contains(Henry)').next().addClass('highlight');  
    $('td:contains(Henry)').parent().find('td:eq(1)')  
        .addClass('highlight').end().find('td:eq(2)')  
        .addClass('highlight');  
});
```

- **Acceso a los elementos nativos DOM**
  - Aunque no suele ser necesario, se puede acceder a los elementos nativos del árbol DOM con el método `get(...)`

```
varmyTag = $('#my-element').get(0).tagName;
```

- Asociar manejadores de eventos a los elementos


```
$(document).ready(function() {  
    $('#switcher-large').on('click', function() {  
        $('body').addClass('large');  
    });  
});
```

```
$(document).ready(function() {  
    $('#switcher-large').on('click', function() {  
        $(this).addClass('selected');  
    });  
});
```

**this** apunta al elemento en el que se ha generado el evento

- Asociar manejadores de eventos a los elementos
  - Los eventos se pueden producir en el elemento seleccionado o en cualquiera de sus descendientes
  - Se dice que el evento burbujea (***bubble***)
  - La función manejadora puede recibir el objeto **event**
    - **event.target**: Objeto que origina el evento
    - **event.preventDefault()**: Evita que el browser realice el comportamiento por defecto
    - **event.stopPropagation()**: Evita que se ejecuten manejadores de los elementos padre. Se para el burbujeo.

```
$('#panel').on('click', function(event) {  
    console.log('Generado en:' + event.target);  
});
```



- Crea una página con campo de texto y un botón
- Cada vez que se pulse el botón, se añadirá el contenido del cuadro de texto a la página



- Se desea implementar una **barra de herramientas** para modificar los **estilos de un documento**

**A Christmas Carol**  
**In Prose, Being a Ghost Story of Christmas**  
by Charles Dickens  
**Preface**  
I HAVE endeavoured in this Ghostly little book, to raise the Ghost of an Idea, which shall not put my readers out of humour with themselves, with each other, with the season, or with me. May it haunt their houses pleasantly, and no one wish to lay it.  
Their faithful Friend and Servant,  
C. D.  
December, 1843.  
**Stave I: Marley's Ghost**  
MARLEY was dead: to begin with. There is no doubt whatever about that. The register of his burial was signed by the clergyman, the clerk, the undertaker, and the chief mourner. Scrooge signed it: and Scrooge's name was good upon 'Change, for anything he chose to put his hand to. Old Marley was as dead as a door-nail.

**Style Switcher**

- Documento de ejemplo (descargar)

### **A Christmas Carol**

#### **In Prose, Being a Ghost Story of Christmas**

by Charles Dickens

##### **Preface**

I HAVE endeavoured in this Ghostly little book, to raise the Ghost of an Idea, which shall not put my readers out of humour with themselves, with each other, with the season, or with me. May it haunt their houses pleasantly, and no one wish to lay it.

Their faithful Friend and Servant,

C. D.

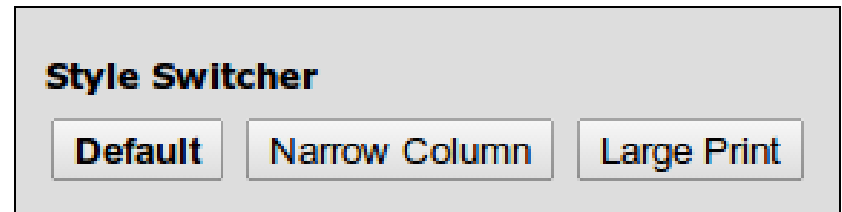
December, 1843.

##### **Stave I: Marley's Ghost**

MARLEY was dead: to begin with. There is no doubt whatever about that. The register of his burial was signed by the clergyman, the clerk, the undertaker, and the chief mourner. Scrooge signed it: and Scrooge's name was good upon 'Change, for anything he chose to put his hand to. Old Marley was as dead as a door-nail.

- Barra de herramientas

```
<div id="switcher" class="switcher">
  <h3>Style Switcher</h3>
  <button id="switcher-default">
    Default
  </button>
  <button id="switcher-narrow">
    Narrow Column
  </button>
  <button id="switcher-large">
    Large Print
  </button>
</div>
```



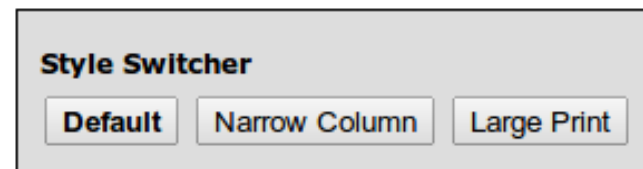
- Barra de herramientas

- Botones

- ▢ **Large print:** Aplicar el estilo CSS "large" al body (y quitar los demás)
    - ▢ **Narrow Column:** Aplicar el estilo CSS "narrow" al body (y quitar los demás)
    - ▢ **Default:** Volver al estilo normal

- Opción seleccionada

- ▢ La **opción seleccionada** debería reflejarse mostrando el texto del botón en **negrita** (estilo "selected")



- Barra de herramientas
  - Al cargar el documento, la barra de herramientas debería aparecer **minimizada** (usando la clase CSS "hidden")
  - Al pulsar en ella, debe cambiar de modo **minimizado** a **normal** (y viceversa)



- Barra de herramientas
  - Las teclas también pueden configurar el estilo:
    - D: Default
    - N: Narrow
    - L: Large



- **Utilidades JavaScript / jQuery necesarias**
  - `$(...).is("button")`: Indica si el elemento es un botón
  - `"sss-www".split('-')`: divide el String en un array de Strings dividiendo por '-'
  - `$(document).keyup(...)`: Se lanza cuando se pulsa una tecla
  - `String.fromCharCode(event.which)`: Devuelve la tecla pulsada como String

- Introducción
- El lenguaje JavaScript
- HTML Interactivo
- **Orientación a objetos avanzada**
  - Clases vs Prototipos
  - Función Constructor
  - Simulación de Clases en JavaScript



- La orientación a objetos en **JavaScript** a primera vista parece **similar** a la de **Java**, pero en esencia es **muy diferente**
- **Clases vs prototipos**
  - La gran **mayoría de los lenguajes** de programación implementan la **POO basada en clases** (Java, C#, C++, Python, Ruby...)
  - Pero JavaScript implementa la **POO basada en prototipos**

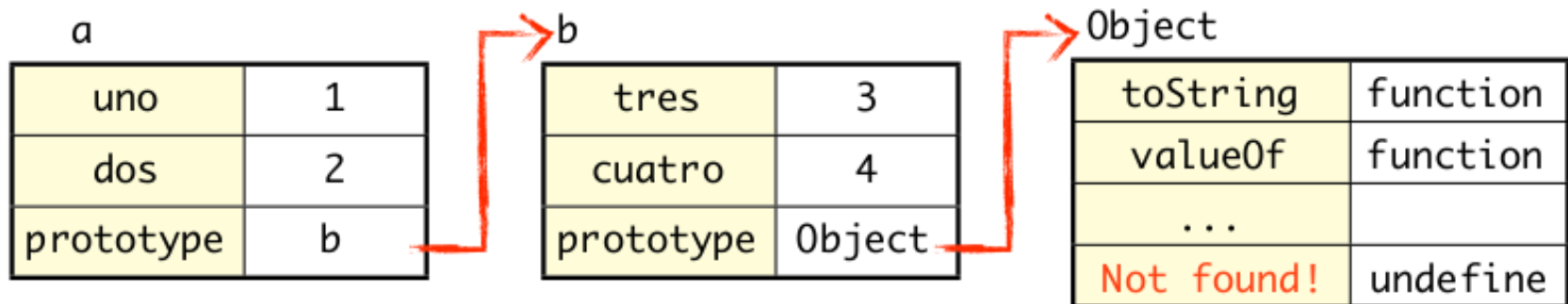
- POO basada en clases (Java)
  - Los objetos son **ejemplares de una clase**
  - La **clase** define los **atributos y métodos** que tendrán los objetos de esa clase
  - Cada objeto se diferencia entre sí por el valor de los atributos
  - Todos los **objetos** de una clase tienen los **mismos atributos (quizás con valores diferentes) y los mismos métodos**

- **POO basada en prototipos (JavaScript)**
  - No existen las **clases**
  - Cada objeto puede tener **cualquier atributo o método**
  - Los objetos se pueden **asociar a otros objetos especiales** (sus **prototipos**)
  - Si se intenta acceder a un método en un objeto y ese método no está en el objeto, **se busca en su prototipo**
  - Si se intenta leer el valor de un atributo en un objeto y ese atributo no está en el objeto, **se busca en su prototipo**

# ORIENTACIÓN A OBJETOS AVANZADA

## Clases vs Prototipos

- Prototipos



```
a.uno;           // Valor 1
a.tres;          // Valor 3 (objeto b)
a.toString()     // Ejecuta el método toString (Object)
```

- **Objetos con los mismos atributos y métodos**
  - En POO es habitual tener varios objetos de **la misma clase**
  - Eso implica que esos objetos tienen los **mismos métodos y atributos**
  - En **JavaScript** se puede conseguir esa funcionalidad si implementamos una **función para crear todos los objetos “similares”** (de la misma “clase”)
  - De esa forma, **no tenemos que repetir** el código de creación del objeto por todo el programa

- Objetos con los mismos atributos y métodos

```
function crearEmpleado(nombre, salario){  
    var empleado = {  
        nombre: nombre,  
        salario: salario,  
        toString: function(){  
            return "N:"+this.nombre+" S:"+this.salario;  
        }  
    };  
    return empleado;  
}
```

- **Objetos con los mismos atributos y métodos**
  - Esa función para crear objetos se utilizaría:

```
var empleado = crearEmpleado("Pepe",700);  
  
//Devuelve 'Nombre:Pepe, Salario:700'  
var texto = empleado.toString();  
  
//Devuelve 700  
var salario = empleado.salario;
```

- **Prototipos**

- En el ejemplo de los **empleados**, cada empleado puede tener **valores diferentes en cada atributo**, pero todos tendrán los **mismos métodos**
- Para optimizar la memoria se puede hacer que todos los objetos empleado usen el mismo **prototipo**, de forma que los métodos estén en un **único objeto**
- Si se quiere **añadir** un nuevo **método** a todos los empleados y todos los empleados están asociadas al un mismo prototipo, entonces basta con **añadir el método al prototipo**



# ORIENTACIÓN A OBJETOS AVANZADA

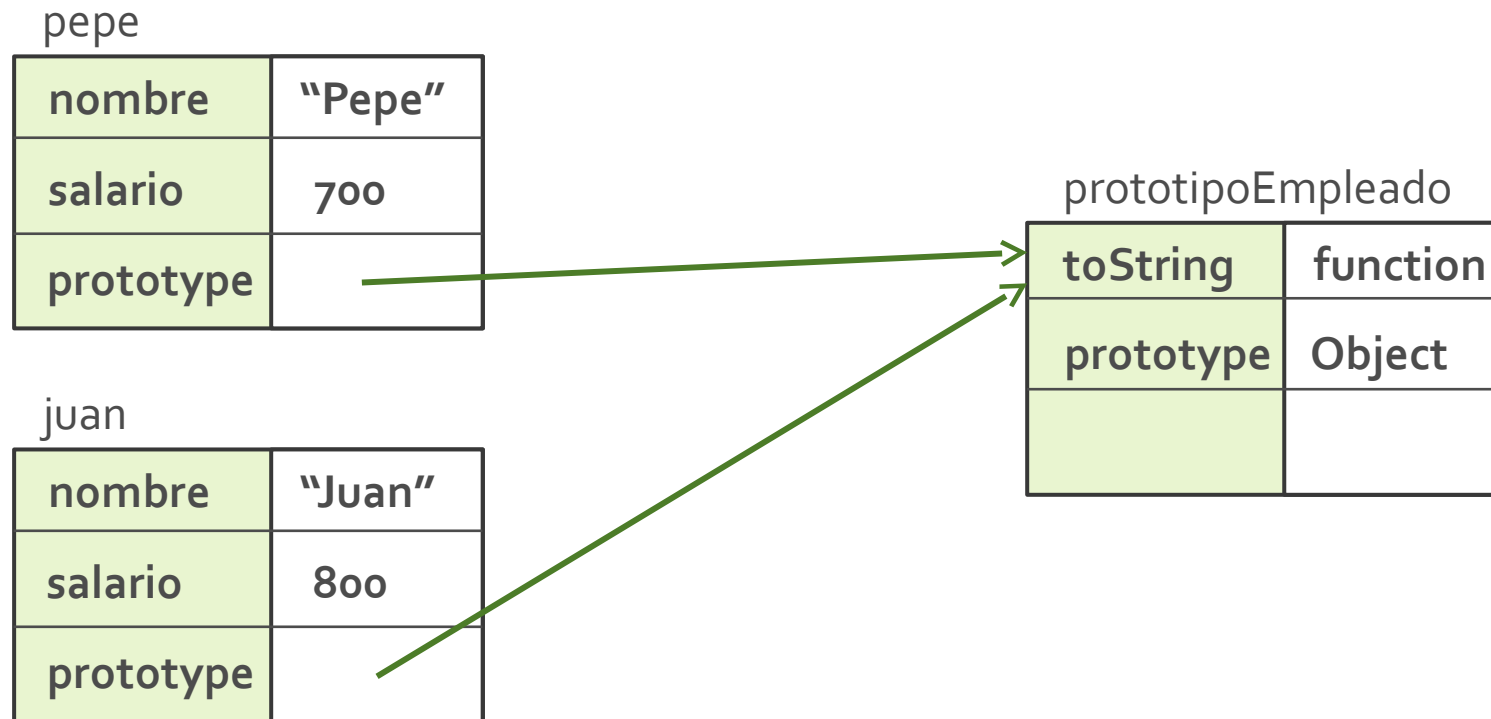
## Clases vs Prototipos

```
var prototipoEmpleado = {  
  toString: function(){  
    return "Nombre:"+this.nombre+", Salario:"+this.salario;  
  }  
};  
  
function newEmpleado(nombre, salario){  
  var empleado = Object.create(prototipoEmpleado);  
  empleado.nombre = nombre;  
  empleado.salario = salario;  
  return empleado;  
}
```

Crea un objeto con un  
prototipo asociado

# ORIENTACIÓN A OBJETOS AVANZADA

## Clases vs Prototipos



```
var pepe = crearEmpleado("Pepe", 700);  
var juan = crearEmpleado("Juan", 800);
```

- **Características de los Prototipos**

- Cada objeto se puede asociar a un único **prototipo**
- Cuando se lee el valor de un **atributo** en un objeto y no existe en dicho objeto, se **busca** en su prototipo y se devuelve su valor.
- Cuando se escribe el valor de un **atributo** en un objeto y no existe en ese objeto, **se crea el atributo en el objeto, no en el prototipo**.
- Cuando se intenta **ejecutar** un método en un objeto y no existe en dicho objeto, se **busca** en su prototipo y se ejecuta.
- A un objeto se le pueden añadir y quitar atributos y métodos en tiempo de ejecución, pero **no puede cambiar de prototipo**.

- **Un prototipo no es una clase**
  - Como estamos acostumbrados a pensar en la POO basada en **clases**, el prototipo de un objeto nos **parece como si fuera una clase**
  - Conviene no olvidar nunca que estamos trabajando con **prototipos**, porque es como funciona realmente JavaScript

- **Función constructor**

- El esquema para trabajar con prototipos es tan habitual que existe una **sintaxis especial en JavaScript para crear objetos asociados a prototipos**
- El operador **new** se puede usar para invocar funciones constructor, especialmente diseñadas para crear **objetos asociados a prototipos**
- La sintaxis de creación de objetos recuerda a Java, pero se implementa **con prototipos en vez de clases**

# ORIENTACIÓN A OBJETOS AVANZADA

## Función constructor

```
function crearEmpleado(nombre, salario){  
    var empleado = Object.create(prototipoEmpleado);  
    empleado.nombre = nombre;  
    empleado.salario = salario;  
    return empleado;  
}
```

```
var empleado = crearEmpleado("Pepe", 700);
```



```
function Empleado(nombre, salario){  
    this.nombre = nombre;  
    this.salario = salario;  
}
```

```
var empleado = new Empleado("Pepe", 700);
```

Cuando se utiliza el **operador new** al **llamar** a una función, se comporta como un **constructor** porque **this** apunta a un **nuevo objeto** que se devuelve al terminar

# ORIENTACIÓN A OBJETOS AVANZADA

## Función constructor

```
function crearEmpleado(nombre, salario){  
    var empleado = Object.create(prototipoEmpleado);  
    empleado.nombre = nombre;  
    empleado.salario = salario;  
    return empleado;  
}
```

```
var empleado = crearEmpleado("Pepe", 700);
```



```
function Empleado(nombre, salario){  
    this.nombre = nombre;  
    this.salario = salario;  
}
```

```
var empleado = Empleado("Pepe", 700);
```

### CUIDADO!!!!

Nunca llames a una función constructor sin el operador new

Pasarán cosas malas (pero no aparecerá un error hasta mucho más adelante)

- ¿Cuál es el prototipo cuando se usa un constructor?
  - El prototipo es el objeto que tenga la función en su atributo **prototype**
  - Para añadir métodos al prototipo, podemos acceder directamente a ese atributo

```
Empleado.prototype.toString = function() {  
    return "Nombre:"+this.nombre+",  
        Salario:"+this.salario;  
};
```



- **Simulación de clases en JavaScript**

- Como hemos visto, JavaScript no tiene clases, tiene **objetos dinámicos y prototipos**
- Un desarrollador Java puede sentirse más cómodo programando **si emula el concepto de clase** usando **objetos y prototipos**
- No obstante, la **sintaxis es un poco confusa** y la **herencia de clases no se consigue de forma directa** (es un poco enrevesada)

- Existen tres formas de **simular** clases en JavaScript:
  - Como hemos visto hasta ahora, usando los **prototipos directamente**
  - Usando el **ámbito de las funciones para crear módulos** (simulados con clousures)
  - Usando **librerías JavaScript** con azúcar sintáctico
- No estudiaremos los módulos ni las librerías por falta de tiempo, pero conviene saber que existen otros enfoques
- Compararemos las **clases Java** con el uso de **prototipos en JavaScript**

# ORIENTACIÓN A OBJETOS AVANZADA

## Simulación de clases en JavaScript

### Clase en Java

```
public class Empleado {  
  
    private String nombre;  
    private double salario;  
  
    public Empleado(String nombre,  
        double salario){  
        this.nombre = nombre;  
        this.salario = salario;  
    }  
  
    public String getNombre(){  
        return nombre;  
    }  
  
    public String toString(){  
        return "Nombre:"+nombre+","  
            + "Salario:"+salario;  
    }  
}
```

### Simulación de clase en JS con prototipos



En JavaScript los atributos no se declaran de forma explícita. Se asignan en la función constructor

```
function Empleado(nombre, salario){  
  
    this.nombre = nombre;  
    this.salario = salario;  
}  
  
Empleado.prototype.getNombre = function(){  
    return nombre;  
};  
  
Empleado.prototype.toString = function(){  
    return "Nombre:"+this.nombre+","  
        + "Salario:"+this.salario;  
};
```

## Creación y uso de objetos en Java

```
Empleado empleado = new Empleado("Juan",800);  
System.out.println( empleado.toString() );
```

## Creación y uso de objetos en JavaScript

```
var empleado = new Empleado("Pepe", 700);  
console.log( empleado.toString() );
```

- **Diferencias Java vs JavaScript**

- En JS todos los atributos son públicos
- En JS todo es dinámico, en cualquier parte del código, en cualquier momento, se puede:
  - ▢ Añadir / Eliminar atributos al objeto
  - ▢ Añadir / Eliminar / Cambiar métodos al objeto
  - ▢ Añadir / Eliminar atributos al prototipo (y todos los objetos que comparten el prototipo lo ven)
  - ▢ Añadir / Eliminar / Cambiar métodos al prototipo (y todos los objetos que comparten el prototipo lo ven)

- Preguntar si un **objeto** se ha creado con una determinada **función constructor** (preguntar por la “**clase**” de un objeto)

```
function procesaObjeto(obj) {  
    if(obj instanceof Empleado) {  
        console.log('Empleado');  
        ...  
    }  
}
```

- Cuidado al usar this

```
function Empleado(nombre, sueldo) {  
    this.nombre = nombre;  
    this.sueldo = sueldo;  
}  
  
Empleado.prototype.alertaAlPulsar(button) {  
  
    button.onclick = function() {  
        alert(...);  
    }  
}
```

¿Cómo accedo a los **atributos** del empleado desde la función?

# ORIENTACIÓN A OBJETOS AVANZADA

## Simulación de clases en JavaScript

- Cuidado al usar this


```
function Empleado(nombre, sueldo) {  
    this.nombre = nombre;  
    this.sueldo = sueldo;  
}  
  
Empleado.prototype.alertaAlPulsar(button) {  
  
    button.onclick = function() {  
        alert(this.nombre + " " + this.sueldo);  
    }  
}
```

**ERROR:** this apunta al objeto en el que se ha generado el evento



- Cuidado al usar this

```
function Empleado(nombre, sueldo) {  
    this.nombre = nombre;  
    this.sueldo = sueldo;  
}  
  
Empleado.prototype.alertaAlPulsar(button) {  
    var that = this;  
    button.onclick = function() {  
        alert(that.nombre+" "+that.sueldo);  
    }  
}
```



Se usa una variable auxiliar,  
**that**, para referenciar al objeto

- **Herencia de clases**

- Se puede emular usando cadenas de prototipos
- En Java:
  - Si un método no está definido en la clase hija, se usa la definición de la clase padre
- En JavaScript:
  - Si un método no está definido en el prototipo directo del objeto, se busca en el **prototipo del prototipo (cadena de prototipos)**

# ORIENTACIÓN A OBJETOS AVANZADA

## Simulación de clases en JavaScript

### Clase hija en Java

```
public class Jefe extends Empleado {  
  
    private String despacho;  
  
    public Jefe(String nombre, double salario,  
        String despacho){  
        super(nombre, salario);  
        this.despacho = despacho;  
    }  
  
    public String toString(){  
        return super.toString()+" Despacho:"+this.despacho;  
    }  
}
```

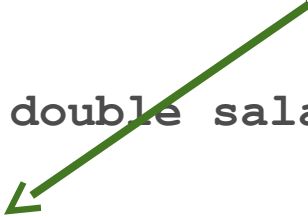
# ORIENTACIÓN A OBJETOS AVANZADA

## Simulación de clases en JavaScript

### Clase hija en Java

```
public class Jefe extends Empleado {  
  
    private String despacho;  
  
    public Jefe(String nombre, double salario,  
                String despacho){  
        super(nombre, salario);  
        this.despacho = despacho;  
    }  
  
    public String toString(){  
        return super.toString()+" Despacho:"+this.despacho;  
    }  
}
```

Llamada al constructor de la clase padre



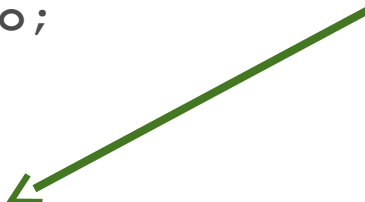
# ORIENTACIÓN A OBJETOS AVANZADA

## Simulación de clases en JavaScript

### Clase hija en Java

```
public class Jefe extends Empleado {  
  
    private String despacho;  
  
    public Jefe(String nombre, double salario,  
                String despacho){  
        super(nombre, salario);  
        this.despacho = despacho;  
    }  
  
    public String toString(){  
        return super.toString()+" Despacho:"+this.despacho;  
    }  
}
```

Llamada al método de la clase padre desde la redefinición del método en la clase hija



# ORIENTACIÓN A OBJETOS AVANZADA

## Simulación de clases en JavaScript

### Simulación de clase hija en JavaScript

```
Jefe.prototype = Object.create(Empleado.prototype);

function Jefe(nombre, salario, despacho){
    Empleado.call(this, nombre, salario);
    this.despacho = despacho;
}


Jefe.prototype.toString = function(){
    return Empleado.prototype.toString.call(this) + "
        Despacho: "+this.despacho;
};
```

## Simulación de clase hija en JavaScript

```
Jefe.prototype = Object.create(Empleado.prototype);
```

```
function Jefe(nombre, salario, despacho){  
    Empleado.call(this, nombre, salario);  
    this.despacho = despacho;  
}
```

```
Jefe.prototype.toString = function(){  
    return Empleado.prototype.toString.call(  
        Despacho: "+this.despacho;  
    };
```




Jefe hereda de Empleado se representa haciendo que el prototipo de Jefe sea el Empleado

## Simulación de clase hija en JavaScript

```
Jefe.prototype = Object.create(Empleado.prototype);  
  
function Jefe(nombre, salario, despacho){  
    Empleado.call(this, nombre, salario);  
    this.despacho = despacho;  
}  
  
Jefe.prototype.toString = function(){  
    return Empleado.prototype.toString.call(this) + "  
        Despacho: "+this.despacho;  
};
```

Llamada al  
constructor de la  
clase padre





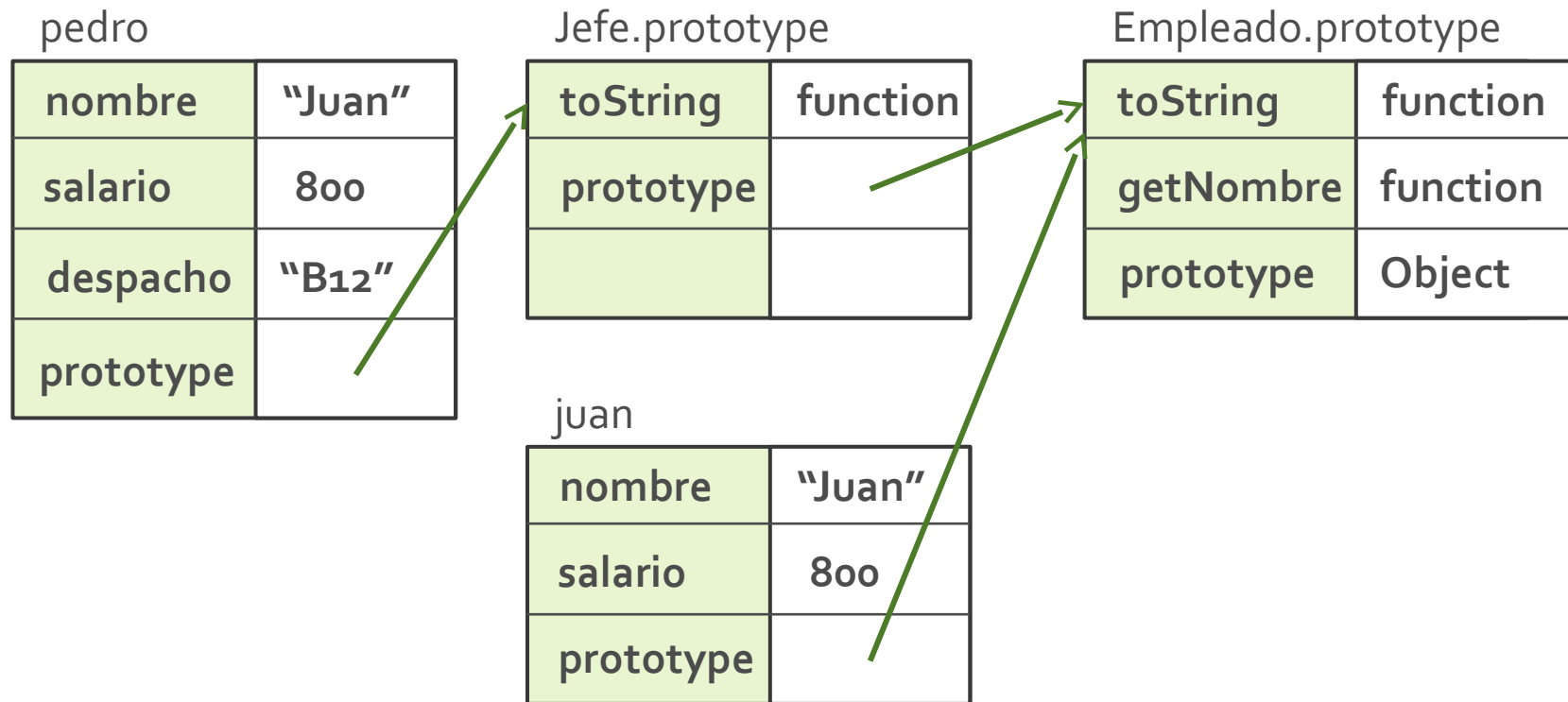
## Simulación de clase hija en JavaScript

```
Jefe.prototype = Object.create(Empleado.prototype);  
  
function Jefe(nombre, salario, despacho){  
    Empleado.call(this, nombre, salario);  
    this.despacho = despacho;  
}  
  
Jefe.prototype.toString = function(){  
    return Empleado.prototype.toString.call(this) + "  
        Despacho: "+this.despacho;  
};
```

Llamada al método de la clase padre desde la redefinición del método en la clase hija

# ORIENTACIÓN A OBJETOS AVANZADA

## Simulación de clases en JavaScript

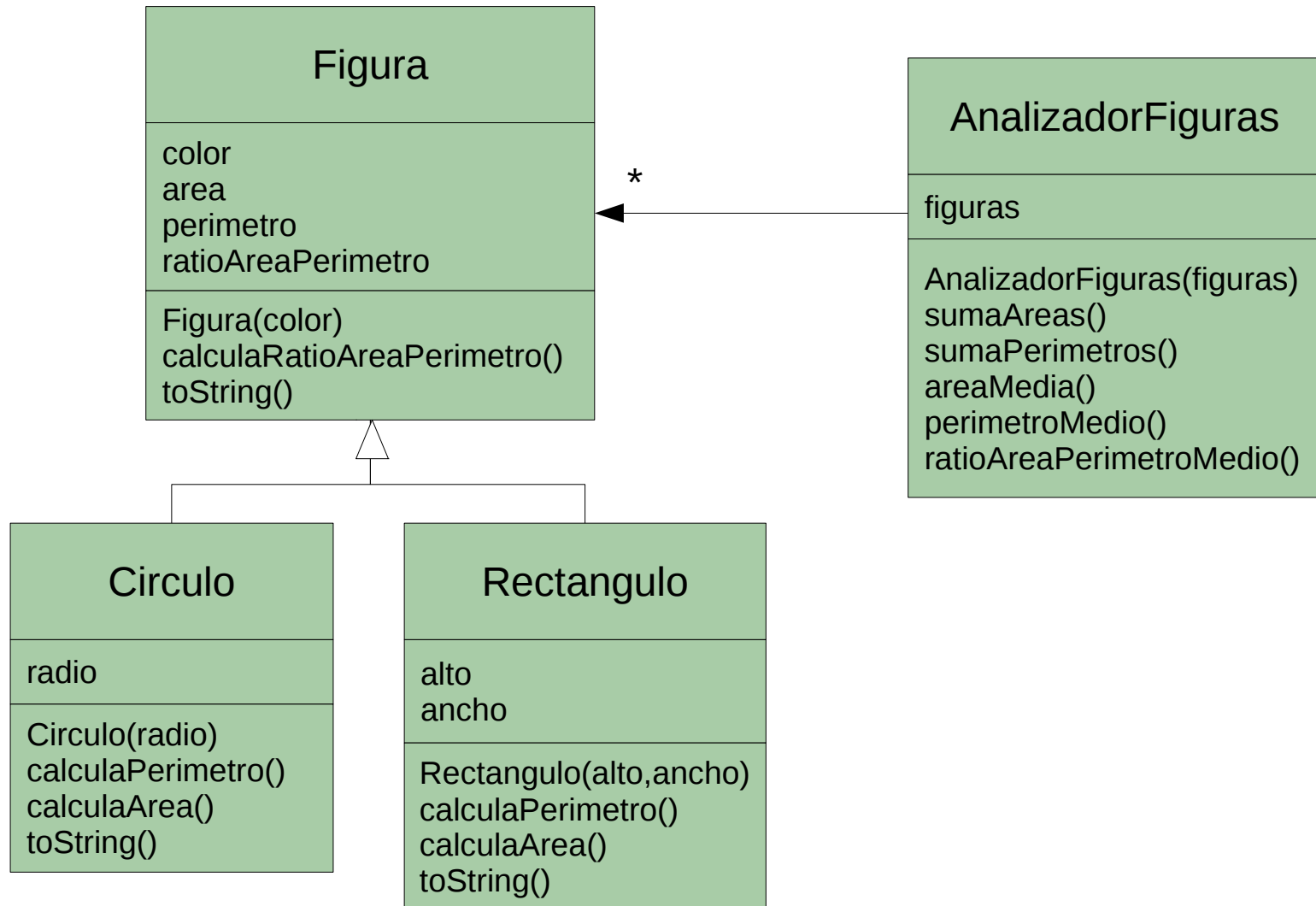


```
var pedro = new Jefe("Pedro", 900, "B12");  
var juan = new Empleado("Juan", 800);
```

- Crear una jerarquía de herencia para representar **figuras geométricas**
- De cada figura se debe conocer:
  - **Color**
  - **Perímetro**
  - **Área**
  - **Ratio entre el área y el perímetro (area / per)**
- Las figuras concretas que tiene que contemplar el programa son **círculos** y **rectángulos**

# ORIENTACIÓN A OBJETOS AVANZADA

## Ejercicio 5



# ORIENTACIÓN A OBJETOS AVANZADA

## Ejercicio 5

- Implementar la clase **AnalizadorFiguras** que permita analizar un array de figuras.
- Los **análisis** serán:
  - Suma total de áreas
  - Suma total de perímetros
  - Área media
  - Perímetro medio
  - Ratio area perímetro medio
- Se deberá crear un **script** que:
  - Construya un array con diferentes figuras de distintos tipos y con distintos valores
  - Ejecute todos los análisis del AnalizadorFiguras
  - Muestre el resultado en la página HTML

- **Círculo**
  - Área:  $\text{Math.PI} * \text{radio} * \text{radio}$
  - Perímetro:  $2 * \text{Math.PI} * \text{radio}$
- **Rectángulo**
  - Área:  $\text{largo} * \text{ancho}$
  - Perímetro:  $2 * \text{ancho} + 2 * \text{largo}$