



Tema 2. Tecnologías del cliente. JavaScript

Programación web

Boni García
Curso 2017/2018

Índice

1. HTML
2. CSS
3. Bootstrap
4. JavaScript
5. jQuery

Índice

1. HTML
2. CSS
3. Bootstrap
4. JavaScript
 - Introducción
 - Características
 - Formas de insertar JavaScript en HTML
 - Sintaxis básica
 - Arrays
 - Sentencias de control
 - Funciones
 - Excepciones
 - *Document Object Model (DOM)*
 - *Browser Object Model (BOM)*
 - Cookies
 - Modo estricto
 - AJAX
 - Orientación a Objetos
 - ECMAScript 6
5. jQuery

Introducción

- **JavaScript** es un lenguaje de programación de alto nivel que se puede ejecutar en los navegadores web
- Surgió en 1995, implementado en el navegador Netscape
- El nombre JavaScript se eligió al nacer en una época en la que Java estaba en auge (inicialmente se llamó Mocha y después LiveScript)
- Como veremos, JavaScript permite incorporar interactividad en el lado cliente de las aplicaciones web:
 - Manipulación de las páginas web (DOM, *Document Object Model*)
 - Peticiones en segundo plano (AJAX, *Asynchronous JavaScript And XML*)
 - ...

Introducción

- El estándar que define el lenguaje JavaScript se llama **ECMAScript**
- Este estándar lo publica la organización de estandarización **ECMA** (*European Computer Manufacturers Association*)
- Las versiones de ECMAScript son:
 - ES5: Versión mínima soportada por los navegadores modernos
 - ES6 (ECMAScript 2015): Incorpora uso de promesas, variables con ámbito...
 - ES7 (ECMAScript 2016): Operador de exponenciación, ...
 - ES8 (ECMAScript 2017): Incorpora uso de generadores...
 - ES.NEXT: Nombre dinámico que se refiere a la siguiente versión en el momento actual

Introducción

- Llamamos JavaScript a la implementación de ECMAScript de los diferentes navegadores

The screenshot shows the ECMAScript 5 compatibility table. The table lists various features and their support status across different browsers and engines. The columns represent different browsers and engines, including es5-shim, Kona 4.14, IE 11, Edge 15, Edge 16, FF 52 ESR, FF 57, FF 58, CH 63, OP 50, SF 10.1, SE 11, and BESK. The rows list features such as Object/array literal extensions, Object static methods, Array methods, String properties and methods, Date methods, Function prototype bind, JSON, Immutable globals, Miscellaneous, and Strict mode. The support status is indicated by colored cells: green for full support, yellow for partial support, and red for no support.

The screenshot shows the ECMAScript 6 compatibility table. The table lists various features and their support status across different browsers and engines. The columns represent different browsers and engines, including Traceur, Babel 6 + core-js, Closure, Type-Script + core-js, es6-shim, Kona 4.14, IE 11, Edge 15, Edge 16, FF 52 ESR, FF 57, FF 58, CH 63, OP 50, SF 10.1, and SE 11. The rows list features such as proper tail calls (tail call optimisation), Syntax (default function parameters, rest parameters, spread / ...operator, object literal extensions, for...of loops, octal and binary literals, template literals), RegExp "y" and "u" flags, destructuring declarations, destructuring assignment, destructuring parameters, Unicode code point escapes, and new.target. The support status is indicated by colored cells: green for full support, yellow for partial support, and red for no support.

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

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

Introducción

- El estándar que define el lenguaje JavaScript se llama **ECMAScript**
- Este estándar lo publica la organización de estandarización **ECMA** (*European Computer Manufacturers Association*)
- Las versiones de ECMAScript son:
 - ES5: Versión mínima soportada por los navegadores modernos
 - ES6 (ECMAScript 2015): Incorpora uso de promesas, variables con ámbito...
 - ES7 (ECMAScript 2016): Operador de exponenciación, ...
 - ES8 (ECMAScript 2017): Incorpora uso de generadores...
 - ES.NEXT: Nombre dinámico que se refiere a la siguiente versión en el momento actual

Características

- **Imperativo y estructurado**
 - Se declaran **variables**
 - Se ejecutan las sentencias **en orden**
 - Dispone de **sentencias de control** de flujo de ejecución
- **Scripting:** Tradicionalmente JavaScript ha sido interpretado en el navegador
 - Esto no es cierto para todos los navegadores
 - Por ejemplo, el motor de JavaScript de Google Chrome (llamado V8) **compila** el código JavaScript a código máquina siguiendo un enfoque JIT (*just-in-time*), esto es, realizando optimizaciones al código compilado en tiempo de ejecución (cosa que un intérprete no puede hacer)

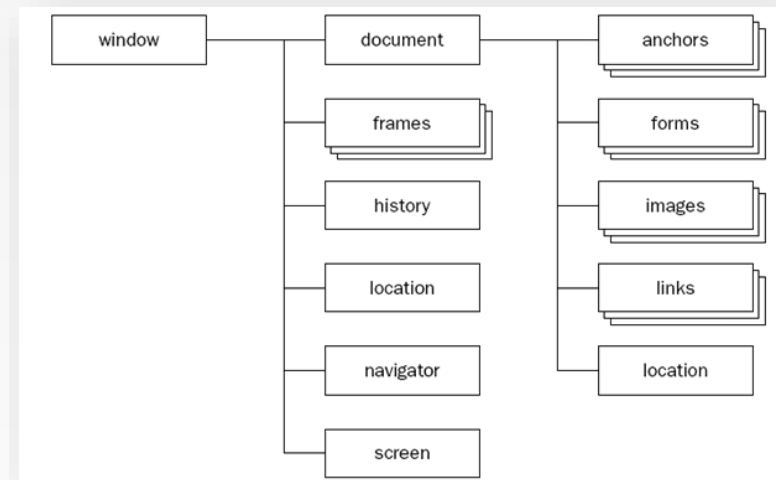


Características

- **Tipado dinámico**
 - Al declarar una variable no es necesario definir su tipo
 - A lo largo de la ejecución del programa una misma variable puede tener valores de diferentes tipos
- **Orientado a objetos**
 - Existe **recolector de basura** para objetos que no se utilizan
 - La orientación a objetos está **basada en prototipos** (se pueden crear objetos, añadir atributos y métodos en tiempo de ejecución)
- **Funcional**
 - Las funciones en JavaScript son elementos de primera clase, o sea, permite declarar **funciones independientes**
 - Esas funciones se pueden declarar en cualquier sitio, asignarse a variables, pasarse como parámetro

Características

- Permite la manipulación del **DOM** y del **BOM**
- DOM (*Document Object Model*)
 - 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

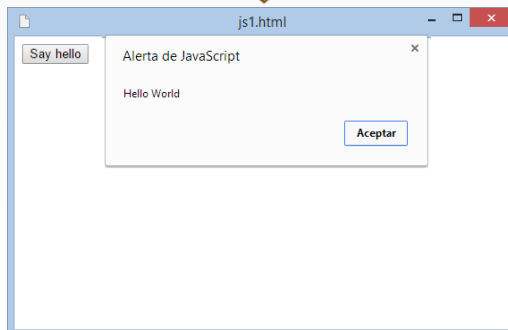


Formas de insertar JavaScript en HTML

1. Incluido en un elemento

- Mediante eventos HTML (*onclick*, ...):

```
<!DOCTYPE html>
<html>
<body>
<button
onclick="alert('Hello
World');">Say
hello</button>
</body>
</html>
```



2. Incluido en la página

- En la sección `head` o `body` con la etiqueta `script`:

```
<!DOCTYPE html>
<html>
<head>
<script>
function hello() {
    alert('Hello World');
}
</script>
</head>
<body>
<button
onclick="hello();">Say
hello</button>
</body>
</html>
```



3. En fichero independiente

- Enlazado con etiqueta `script` en sección `head` o `body`:

```
<!DOCTYPE html>
<html>
<head>
<script
src="script.js"></script>
</head>
<body>
<button
onclick="hello();">Say
hello</button>
</body>
</html>
```

script.js

```
function hello() {
    alert('Hello World');
}
```



Sintaxis básica

■ Variables:

- Es un lenguaje con tipado **dinámico** (las variables se declaran con `var` pero no se indica el tipo):

```
var count = 10;  
var found = false;  
var name = 'John';
```

- Tipos de datos:

- `Number` (números enteros y reales de cualquier precisión)
 - `String` (Cadenas de caracteres, separadas por comillas simples o dobles)
 - `Boolean` (`true` o `false`)
- La variable tiene como ámbito la función, no por bloque
 - Si no se inicializan, las variables tienen el valor `undefined`. Valores especiales:
 - Las variables pueden además tomar el valor `null` (variable definida pero con estado nulo)
 - Valor `NaN` (“*not a number*”, valor especial que dictamina que una variable no es numérica)

Sintaxis básica

Operadores:

Similares a Java:

- Aritméticos: + - * / %
- Comparación números: < > <= >=
- Lógicos: && || !
- Comparativo: ? : (operador Elvis)
- Modificación: ++ --
- Asignación: = += -= *= /= %=

Específico de JavaScript:

- Igual (valor y tipo): ===
- Distinto (valor y tipo): !==

```
var x = 5;
console.info(x == "5");
console.info(x == 5);
console.info(x === "5");
console.info(x === 5);
console.info(x != "5");
console.info(x != 5);
console.info(x !== "5");
console.info(x !== 5);
```



```
Console Search Emulation Rendering
<top frame> Preserve log
true
true
false
true
false
false
true
false
```

Sintaxis básica

- Comentarios (una línea y multilínea):

```
// Comment inline  
  
/*  
 * Comment multi-line  
 */
```

- Delimitadores:
 - De bloque { }
 - De sentencia ; (opcional en JavaScript)
- Para escribir en la consola del navegador usamos el objeto `console`:

```
console.log('text');  
console.info('text');  
console.warn('text');  
console.error('text');
```

Arrays

- Características iguales a Java:
 - El acceso para lectura o escritura es con []
 - Tienen la propiedad `length`
 - Empiezan por 0
 - La asignación de variables no copia el array, las variables apuntan al mismo objeto
- Características diferentes a Java:
 - Los literales son con [] en vez de { }
 - No se pone `new` en el literal
 - Pueden mezclar valores de varios tipos
 - El acceso a un elemento fuera de los límites es `undefined`

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

Arrays

- Los arrays en JavaScript se comportan como listas dinámicas:
 - 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
 - El operador `delete` borra un elemento (pero deja el hueco)
 - 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.

```
var numbers = [ 'zero', 'one', 'two', 'three' ];
console.info(numbers[0]);
console.info(numbers);
console.info(numbers.length);
delete numbers[2];
console.info(numbers);
console.info(numbers.length);
numbers.splice(2, 1);
console.info(numbers);
console.info(numbers.length);
```



```
Console | Search | Emulation | Rendering
<top frame> | Preserve log
zero
["zero", "one", "two", "three"]
4
["zero", "one", 3: "three"]
4
["zero", "one", "three"]
3
>
```


Sentencias de control

- Bloques de sentencias
 - Con llaves { }
 - Las variables no desaparecen al terminar en bloque
- Sentencia condicional: `if`
 - Sintaxis como en Java
 - La expresión no tiene que devolver un valor `boolean`
 - Se considera falso: `false`, `null`, `undefined`, "" (cadena vacía), 0, NaN

Sentencias de control

- Sentencias `switch`, `while` y `do`
 - Sintaxis y semántica como en Java
- Sentencia `for`
 - `for(init; expr; inc)`
 - En ES5 la variable se declara fuera del `for`
 - No existe `continue`, pero sí `break`
- Sentencia `return`
 - Rompe el flujo y devuelve

Funciones

- Se pueden declarar con nombre:

```
function func(param) {  
    console.log(param);  
}  
  
func(4); // Print 4 in the console
```

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

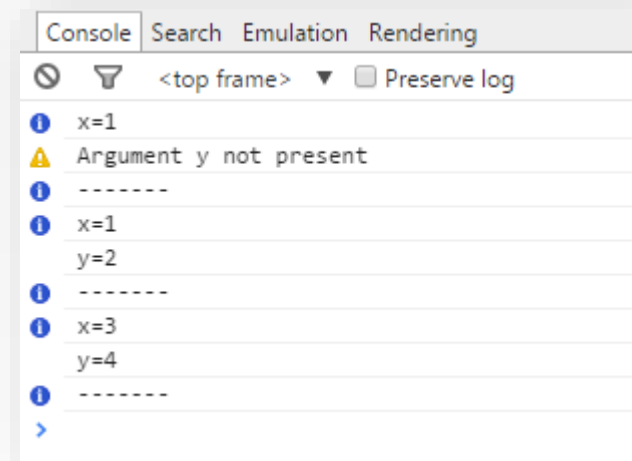
```
var func2 = function(param) {  
    console.log(param);  
}  
  
func2(5); // Print 5 in the console
```

Funciones

- El número de parámetros al invocar una JavaScript no tiene por qué coincidir con el número de parámetros que acepta la función
 - Si se pasan menos parámetros, los demás se pasan como `undefined`
 - Si se pasan más, se ignoran

```
function myFunction(x, y) {
  console.info("x=" + x);
  if (!y) {
    console.warn("Argument y not present");
  } else {
    console.log("y=" + y);
  }
  console.info("-----");
}

myFunction(1);
myFunction(1, 2);
myFunction(3, 4, 5);
```

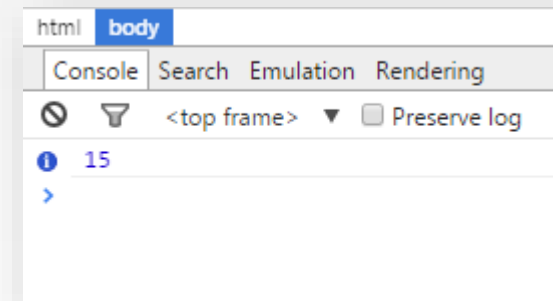


Funciones

- Todas las funciones cuentan con el objeto `arguments`, que contiene el array de los argumentos con la que fue invocada una función:

```
function sumAll() {
  var i, sum = 0;
  for (i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}

console.info(sumAll(1, 2, 3, 4, 5));
```



- Se pueden declarar funciones en el cuerpo de otras funciones:

```
var myFunction1 = function(node) {
  node.onclick = function(e) {
    alert("Alert");
  }
}
```

Excepciones

- Existe un bloque con `try catch finally`
- El operador `throw` eleva la excepción
- A diferencia de Java, se puede lanzar cualquier objeto como excepción

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

Document Object Model (DOM)

- El *Document Object Model* (DOM) es la API para manipular documentos HTML desde JavaScript
- Cuando se carga una página HTML en un navegador, se convierte en un objeto tipo `document`
- Todos los elementos HTML están representado por el objeto `element`
- Acceso a un elemento del documento usando el atributo `id` (el atributo `id` debería ser único en la página. Si no lo fuese, `getElementById` devuelve el primer elemento con dicho `id`):

```
var obj1 = document.getElementById("objId");
```

- Acceso a un elemento del documento por clase CSS (si hay varios elementos, devuelve el primero que encuentra):

```
var obj2 = document.querySelector(".mycssclass");
```

Document Object Model (DOM)

- Acceso a un conjunto de elementos del documento:

- Por el atributo `name`

```
var objArray = document.getElementsByName("objName");
```

- Por el atributo `class`

```
var liArray = document.getElementsByTagName("li");
```

- Por el tipo de etiqueta

```
var classArray = document.getElementsByClassName("myclass");
```

- Modificación de elementos a través del DOM:

- Modificar el contenido HTML de un elemento:

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

- Cambiar el estilo CSS de un elemento:

```
var element = document.getElementById("img1");  
element.style.borderWidth = "3px";
```


Document Object Model (DOM)

- Los eventos HTML pueden ser manejados en JavaScript:

Tipo	Evento	Ocurrencia
Ratón	<code>onclick</code>	Al hacer click con el ratón
	<code>ondblclick</code>	Al hacer doble click con el ratón
	<code>onmouseover</code>	Al pasar por encima el cursor
Teclado	<code>onkeydown</code>	Al presionar una tecla
Página	<code>onload</code>	Al empezar a cargar una página
	<code>onbeforeunload</code>	Justo antes de abandonar una página
Formulario	<code>onchange</code>	Cuando un campo de formulario cambia de valor
	<code>onsubmit</code>	Justo antes de enviar un formulario al servidor

- Más eventos: http://www.w3schools.com/jsref/dom_obj_event.asp

Document Object Model (DOM)

- Ejemplo de captura de eventos (validación de un formulario):

```
<form id="myForm" action="/processForm" method="post">  
  <input type="text" name="txt" id="txt">  
  <input type="submit">  
</form>
```

```
<script>  
window.onload = function() {  
  var form = document.getElementById("myForm");  
  form.addEventListener("submit", function(event) {  
    var txt = document.getElementById("txt");  
    if (!txt.value) {  
      alert("You should provide some value to text input");  
      txt.focus();  
      return event.preventDefault();  
    }  
  });  
}  
</script>
```

Browser Object Model (BOM)

- El objeto `window` representa una ventana/pestaña del navegador
- Algunas propiedades importantes:
 - `document` : Acceso al DOM
 - `history` : Historial de navegación
 - `location`: URL de la página
 - `console`: Consola del navegador
- Algunos métodos importantes:
 - `alert()` : Genera un cuadro de diálogo de tipo alerta
 - `confirm()` : Genera un cuadro de diálogo de tipo confirmación (“aceptar-cancelar”)
 - `open()` : Navega hacia una URL

Cookies

- Las cookies son piezas de información enviadas por una aplicación web y almacenada en la caché del cliente (navegador)
- Están formadas por clave=valor
- Desde mayo de 2011 una normativa europea obliga a todos los sitios webs alojados en Europa a informar del uso de cookies a sus usuarios
 - Más información en: <http://www.cookie-law.org/the-cookie-law/>
- Las cookies se pueden gestionar desde JavaScript mediante la propiedad `cookie` del objeto `document`

Modo estricto

- Usar el modo estricto en ES5 significa optar explícitamente por una variante restringida de JavaScript
 - Atrapa errores comunes de programación, lanzando excepciones
 - Deshabilita algunas características que no son recomendables
- El modo estricto se invoca mediante el siguiente comando:

```
"use strict";
```

- Algunos ejemplos de modo estricto:
 - Las variables tienen que ser declaradas para ser utilizadas
 - Las palabras reservadas no pueden utilizarse como identificadores
 - ...

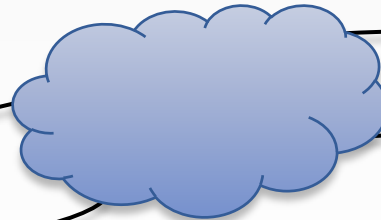
https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Modo_estricto

AJAX

- **AJAX** (*Asynchronous JavaScript And XML*), es una técnica que nos permite hacer peticiones en segundo plano (sin recargar la página) a un servidor web
- En los navegadores modernos, se implementa a través del objeto XMLHttpRequest

Cliente

1. Creación de objeto XMLHttpRequest
2. Envío de petición al servidor
5. Procesado de la respuesta



Servidor

3. Procesado de petición
4. Envío de respuesta al cliente

AJAX

- Constructor del objeto XMLHttpRequest:

```
var xhr = new XMLHttpRequest();
```

- Métodos importantes del objeto XMLHttpRequest:

```
xhr.open(method, url, async); (especifica la petición)
```

- `method` puede ser "GET" o "POST"
- `url` es la ruta en el servidor destino
- `async` determina el tipo de petición: síncrona (`false`) o asíncrona (`true`)

```
xhr.send(); (envía la petición)
```

```
xhr.abort(); (cancela la petición actual)
```

AJAX

- **Propiedades importantes del objeto XMLHttpRequest:**

`xhr.onreadystatechange` (define la función que atiende a la respuesta (*callback*))

`xhr.readyState` (especifica el estado de la petición a nivel AJAX)

- 0 petición no inicializada
- 1 conexión establecida con el servidor
- 2 petición recibida
- 3 petición en proceso
- 4 petición finalizada y respuesta lista

`xhr.status` (especifica el estado de la petición a nivel HTTP)

- 200 ok
- 403 prohibido
- 404 no encontrado

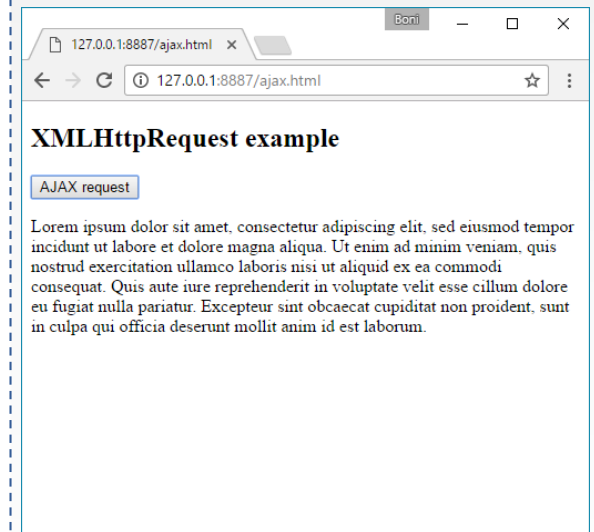
AJAX

```
<!DOCTYPE html>
<html>

<body>
  <h2>XMLHttpRequest example</h2>
  <button type="button" id="mybutton">AJAX request</button>
  <p id="content"></p>
</body>

<script>
  var button = document.getElementById("mybutton");
  button.addEventListener("click", function () {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function () {
      if (this.readyState == 4 && this.status == 200) {
        document.getElementById("content").innerHTML +=
          this.responseText + "<br>";
      }
    };
    xhr.open("GET", "info.txt", true);
    xhr.send();
  });
</script>

</html>
```



Fork me on GitHub

Orientación a objetos

Clases vs prototipos

- La mayoría de lenguajes (Java, C#, C++, Python...) implementan **POO basada en clases**
- JavaScript implementa la **POO basada en prototipos**

POO basada en clases

- Los objetos son instancias de una clase
- La clase se utiliza para definir las propiedades 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 métodos

POO basada en prototipos

- No existen las clases pero sí los **prototipos**
- Se le pueden **añadir** y **borrar** atributos y métodos en cualquier momento
- La herencia se realiza mediante **cadena de prototipos**

Orientación a objetos

Creación de objetos

- En JavaScript tres formas principales de crear un objeto:
 1. Usando un objeto literal
 2. Creando un objeto `Object` con la palabra reservada `new`
 3. Definiendo un constructor y usando la palabra reservada `new`

Orientación a objetos

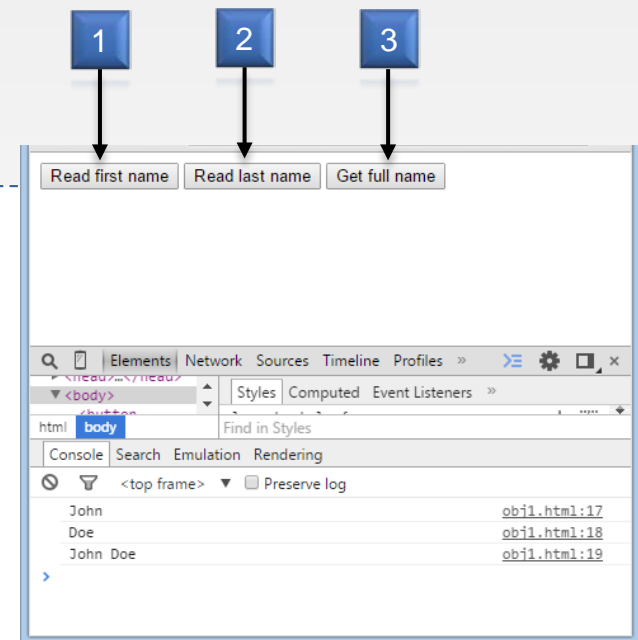
Creación de objetos

1. Usando un objeto literal

```

<!DOCTYPE html>
<html>
<head>
<script>
var person = {
  firstName : "John",
  lastName  : "Doe",
  fullName  : function() {
    return this.firstName + " " + this.lastName;
  }
};
</script>
</head>
<body>
<button onclick="console.log(person.firstName);">Read first name</button>
<button onclick="console.log(person['lastName']);">Read last name</button>
<button onclick="console.log(person.fullName());">Get full name</button>
</body>
</html>

```



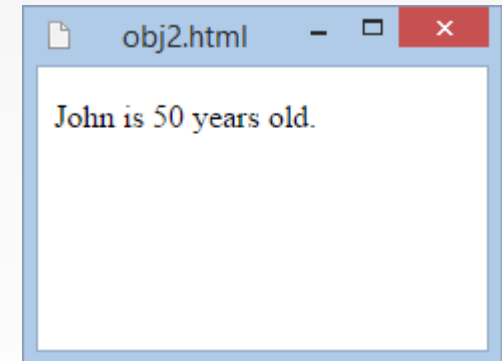
Orientación a objetos

Creación de objetos

2. Creando un objeto `Object` con la palabra reservada `new`

```
<!DOCTYPE html>
<html>
<head>
<script>
window.onload = function() {
    var person = new Object();
    person.firstName = "John";
    person.lastName = "Doe";
    person.age = 50;
    person.eyeColor = "blue";

    document.getElementById("demo").innerHTML = person.firstName
        + " is " + person.age + " years old."
}
</script>
</head>
<body>
<p id="demo" ></p>
</body>
</html>
```



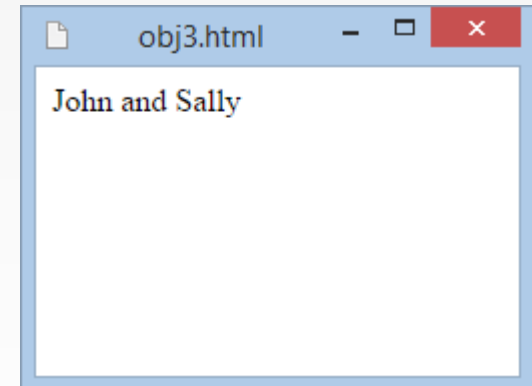
Orientación a objetos

Creación de objetos

3. Definiendo un constructor y usando la palabra reservada `new`

```
<!DOCTYPE html>
<html>
<head>
<script>
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}
var man = new Person("John", "Doe", 50, "blue");
var women = new Person("Sally", "Rally", 48, "green");

document.write(man.firstName + " and " + women.firstName);
</script>
</head>
<body>
</body>
</html>
```

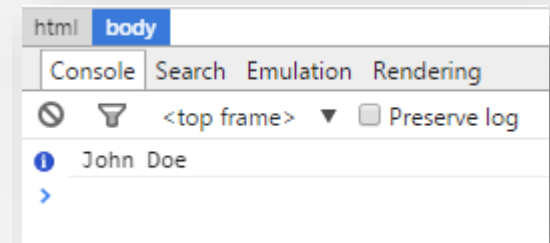


Orientación a objetos

Propiedades de objetos

■ Acceso:

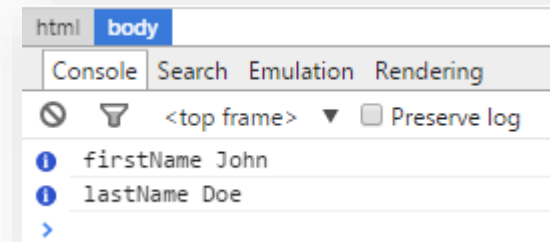
```
var person = {
  firstName : "John",
  lastName  : "Doe"
};
console.info(person.firstName + " " + person["lastName"]);
```



■ Acceso mediante bucle `for-in`:

```
var person = {
  firstName : "John",
  lastName  : "Doe"
};

var i;
for (i in person) {
  console.info(i + " " + person[i]);
}
```

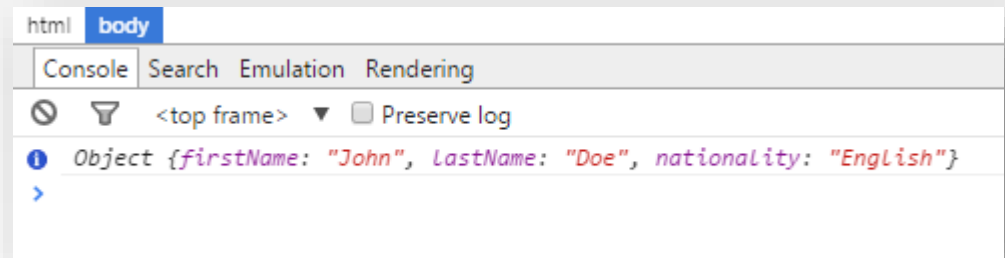


Orientación a objetos

Propiedades de objetos

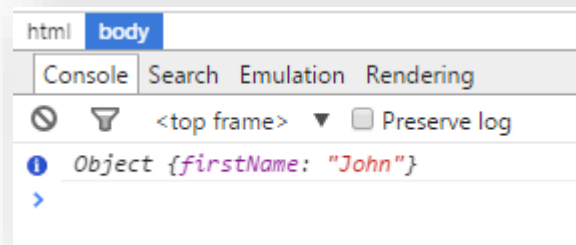
■ Añadir nuevas propiedades:

```
var person = {
  firstName : "John",
  lastName : "Doe"
};
person.nationality = "English";
console.info(person);
```



■ Eliminar propiedades existentes:

```
var person = {
  firstName : "John",
  lastName : "Doe"
};
delete person.lastName;
console.info(person);
```



Orientación a objetos

Métodos de objeto

- Se pueden modificar los métodos de un objeto dinámicamente

```

<!DOCTYPE html>
<html>
<head>
<script>
var person = {
  firstName : "John", lastName : "Doe",
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
function modifyMethod() {
  person.fullName = function() {
    return this.firstName;
  }
}
function deleteMethod() {
  delete person.fullName;
}
</script>
</head>
<body>
<button onclick="console.log(person.fullName());">Full name</button>
<button onclick="modifyMethod();">Modify method</button>
<button onclick="deleteMethod();">Delete method</button>
<button onclick="console.log(person);">Person</button>
</body>
</html>

```

The screenshot illustrates the dynamic modification of a JavaScript object's method. The browser interface shows four buttons: 'Full name', 'Modify method', 'Delete method', and 'Person'. Above these buttons are numbered blue boxes (1-6) with arrows pointing to the buttons. The console shows the initial state of the 'person' object and the state after the 'Person' button is clicked, where the 'fullName' property has been deleted.

Orientación a objetos

Prototipos en JavaScript

- La POO en JavaScript está basada en prototipos
- Todos los objetos en JavaScript **heredan** propiedades y métodos de su prototipo
- La forma estándar de crear un prototipo es mediante su **constructor**
- Hasta ahora hemos visto como añadir/eliminar dinámicamente propiedades y métodos de un objeto, pero también se puede modificar dinámicamente el prototipo

```
function person(first, last, age, eyecolor) {  
  this.firstName = first;  
  this.lastName = last;  
  this.age = age;  
  this.eyeColor = eyecolor;  
}  
person.prototype.name = function() {  
  return this.firstName + " " + this.lastName;  
};
```

Orientación a objetos

Palabra clave `this`

- La palabra clave `this` tiene un comportamiento diferente al de otros lenguajes
- El valor de `this` está determinado por cómo se llama a la función:
 1. En el contexto de ejecución global (fuera de cualquier función), `this` se refiere al objeto global
 2. Dentro de una función/objeto, `this` se refiere al llamante
 3. Dentro de un constructor, el valor de `this` se refiere al objeto creado

<https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/this>

Orientación a objetos

Palabra clave `this`

1. En el contexto de ejecución global (fuera de cualquier función), `this` se refiere al objeto global
2. Dentro de una función/objeto, `this` se refiere al llamante
3. Dentro de un constructor, el valor de `this` se refiere al objeto creado

```

console.log("1 " + this); // Window

function f1() {
  return this;
}
console.log("2.a " + f1()); // Window

function f2() {
  "use strict";
  return this;
}
console.log("2.b " + f2()); // undefined
console.log("2.c " + window.f2()); // Window

var foo = {
  baz : function() {
    console.log("2.d " + this);
  }
}
foo.baz(); // Object

function myObject() {
  this.a = 37;
}
var o = new myObject();
console.log("3 " + o.a); // 37

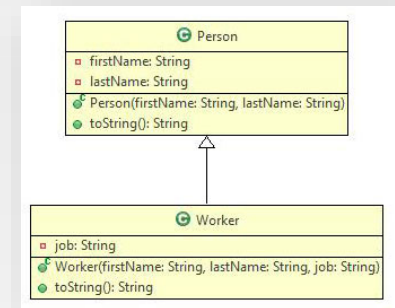
```

1 [object Window]
2.a [object Window]
2.b undefined
2.c [object Window]
2.d [object Object]
3 37

Orientación a objetos

Herencia

- La herencia en JavaScript se hace usando **cadena de prototipos**
- Vamos a analizarlo mediante un ejemplo. Sean las clases Java:



```

public class Person {

    private String firstName;
    private String lastName;

    public Person(String firstName, String
lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName;
    }
}
  
```

```

public class Worker extends Person {

    private String job;

    public Worker(String firstName, String
lastName, String job) {
        super(firstName, lastName);
        this.job = job;
    }

    @Override
    public String toString() {
        return super.toString() + " - Job: "
+ this.job;
    }
}
  
```

Orientación a objetos

Herencia

- El equivalente del ejemplo anterior pero en JavaScript sería:

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype.toString = function() {
  return this.firstName + " "
    + this.lastName;
};
```

```
function Worker(firstName, lastName, job) {
  Person.call(this, firstName, lastName);
  this.job = job;
}

Worker.prototype = new Person();

Worker.prototype.toString = function() {
  return Person.prototype.toString.call(this)
    + " - Job: " + this.job;
};
```

La función `call` permite llamar a un método de un objeto sobrescribiendo el valor de `this`

ECMAScript 6

Ámbito de bloque

- En ES5 sólo hay 2 ámbitos para las variables:
 - Global (definido fuera de todas las funciones)
 - Función (definido dentro de una función)
- ES6 permite la definición de variables con ámbito de **bloque**, definido variables mediante la palabra reservada **let**
- **let** no sustituye a **var** (ambas formas de definir variables coexisten en ES6)

```
for (let i = 0; i < 5; i += 1) {  
  console.log(i);  
}  
console.log(i);
```

```
0  
1  
2  
3  
4  
Uncaught ReferenceError: i is not defined  
at samples.html:12
```

<http://es6-features.org/>

ECMAScript 6

Constantes

- ES6 permite definir **constantes** mediante la palabra reservada **const**

```
const foo = 1;
```

- A diferencia de **let** y **var**, las constantes requieren ser inicializadas en la misma declaración

```
const foo;
```

```
✖ ▶ Uncaught SyntaxError: Missing initializer in const declaration
```

```
const foo = 1;  
foo = 2;
```

```
✖ ▶ Uncaught TypeError: Assignment to constant variable.  
at samples.html:10
```


ECMAScript 6

Cadenas multi-línea

- ES6 permite definir **cadenas multi-línea** usando el carácter ` (“acento grave”, *grave accent*)

```
let multi = `hello  
ES6  
world`;  
  
console.log(multi);
```

```
hello  
ES6  
world  
>
```

```
let name = 'pepito';  
  
let multi2 = `hello  
ES6  
world  
my  
name  
is  
${name}  
`;  
  
console.log(multi2);
```

```
hello  
ES6  
world  
my  
name  
is  
pepito  
>
```

Con este tipo de cadenas podemos además sustituir valores mediante el operador `${}` (*templating*)

ECMAScript 6

Sintaxis de flecha gorda

- En JavaScript las funciones son ciudadanos de primera clase, con lo cual se pueden pasar como argumentos (funciones anónimas)

```
setTimeout(function() {  
  console.log("setTimeout called");  
}, 1000);
```

- Esto se puede simplificar en ES6 mediante lo que se conoce como “sintaxis de flecha gorda” (*fat arrow syntax*)

```
setTimeout(() => console.log("setTimeout called"), 1000);
```

- Con argumentos:

```
let add = function(a,b) {  
  return a + b;  
};
```



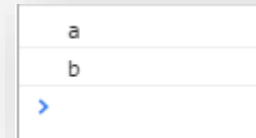
```
let add = (a,b) => a + b;
```

ECMAScript 6

Bucle for-of

- En ES5 disponemos del bucle **for-in** para iterar las propiedades de un objeto

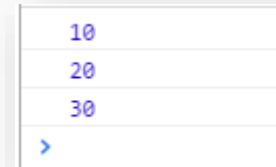
```
let obj = {a:1, b:2};  
for (let prop in obj) {  
  console.log(prop);  
}
```



```
a  
b  
>
```

- En ES6, además disponemos del bucle **for-of** para iterar los valores de un array

```
let array = [10, 20, 30];  
for (let value of array) {  
  console.log(value);  
}
```



```
10  
20  
30  
>
```

ECMAScript 6

Mapas y conjuntos

- En ES6 disponemos del tipo Map (para manipular mapas) y Set (para manipular conjuntos):

```
let map = new Map()
  .set("A", 1)
  .set("B", 2)
  .set("C", 3);

console.log(map.get("A"));
console.log(map.has("A"));
console.log(map.delete("A"));
console.log(map.size);
map.clear();
console.log(map.size);
```

```
1
true
true
2
0
>
```

```
let set = new Set(['APPLE', 'ORANGE', 'MANGO']);

console.log(set.has("APPLE"));
console.log(set.add("BANANA"));
console.log(set.delete("APPLE"));
console.log(set.size);
set.clear();
console.log(set.size);
```

```
true
▶ Set(4) {"APPLE", "ORANGE", "MANGO", "BANANA"}
true
3
0
>
```

ECMAScript 6

Promesas

- Una operación síncrona es bloqueante (se espera hasta que acaba). Por el contrario, las operaciones asíncronas no son bloqueantes
- Las operaciones asíncronas se suelen gestionar mediante *callbacks*
→ Problema: excesivo anidamiento (*callback hell*)

```
getData(function(a){
  getMoreData(a, function(b){
    getMoreData(b, function(c){
      getMoreData(c, function(d){
        getMoreData(d, function(e){
          ...
        });
      });
    });
  });
});
```

ECMAScript 6

Promesas

- ES6 proporciona un mecanismo de **promesas** para gestionar operaciones asíncronas

Usamos `setTimeout()` para simular código asíncrono. En la vida real, probablemente usaríamos algo como Ajax o una llamada a API REST

```
let error = false;
let myPromise = new Promise((resolve, reject) => {
  setTimeout(function() {
    if (error)
      reject("Failure!");
    else
      resolve("Sucess!");
  }, 1000);
});

myPromise.then((successMessage) => {
  console.log("The sucess message is: " + successMessage);
});

myPromise.catch((errorMessage) => {
  console.log("The error message is: " + errorMessage);
});
```

Llamamos a `resolve()` cuando lo que estábamos haciendo finaliza con éxito, y `reject()` cuando falla

ECMAScript 6

Creación de objetos

- ES6 proporciona un mecanismo de creación de **objetos** usando una sintaxis más próxima al patrón clásico de orientación a objetos (como Java o C++) en lugar de la cadena de prototipos

Constructor y
definición de
propiedades de
clase

Métodos de clase

Instanciación de
objeto

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  name() {
    return `${this.firstName} ${this.lastName}`;
  }

  whoAreYou() {
    return `Hi I'm ${this.name()}`;
  }
}

let john = new Person("John", "Doe");
console.log(john.whoAreYou());
```

Hi I'm John Doe



ECMAScript 6

Creación de objetos

- ES6 proporciona un mecanismo de creación de **objetos** usando una sintaxis más próxima al patrón clásico de orientación a objetos (como Java o C++) en lugar de la cadena de prototipos

La herencia se realizan usando la palabra reservada **extends**

```
class Student extends Person {  
  
  constructor(firstName, lastName, course) {  
    super(firstName, lastName);  
    this.course = course;  
  }  
  
  whoAreYou() {  
    return `${super.whoAreYou()} and I'm studying ${this.course}`;  
  }  
}  
  
let michael = new Student("Michael", "Smith", "Web programming");  
console.log(michael.whoAreYou());
```

```
Hi I'm Michael Smith and I'm studying Web programming  
>
```