



CARLOS AZAUSTRE



APRENDIENDO JAVASCRIPT

Aprende las bases del lenguaje web más demandado.
Desde cero hasta ECMAScript 6.

Table of Contents

Introduction	0
Sobre éste libro	1
Historia de JavaScript	2
Orígenes	2.1
Node.js	2.2
Tipos de variables	3
Definición	3.1
Tipos	3.2
Operadores	4
Operadores aritméticos	4.1
Operador typeof	4.2
Operadores booleanos	4.3
Operadores lógicos	4.4
Condicionales	5
Asignación condicional	5.1
Sentencia IF	5.2
Sentencia SWITCH	5.3
Clases Core y Módulos de JavaScript	6
Object	6.1
Number	6.2
Array	6.3
String	6.4
Funciones	7
Parámetros por defecto	7.1
Ámbitos de una función	7.2
Clousures	7.3
Funciones como clases	7.4

Clases en ECMAScript6	7.5
Bucles	8
Bucle While	8.1
Bucle Do/While	8.2
Bucle For	8.3
Bucle ForEach	8.4
Bucle ForIn	8.5
JSON	9
AJAX	10
XMLHttpRequest	10.1
Fetch en ECMAScript6	10.2
Eventos	11
Manejando eventos	11.1
Propagación de eventos	11.2
Patrón PubSub	11.3
Patrón PubSub con Datos	11.3.1
Websockets	11.4
Promesas	12
DOM Document Object Model	13
¿Qué trae nuevo ECMAScript 6?	14
Recapitulación	15

Apreniendo JavaScript

Aprende las bases del lenguaje web más demandado. Desde cero hasta ECMAScript 6.

Copyright © 2015-2016 Carlos Azaustre por la obra y la edición.

- 1ª Edición: **Abril 2016**
 - Versión 1.1: 4 de Abril de 2016

Agradecimientos especiales a **Miguel Ruiz**, **Leonel Contreras** y **Borja Campina** por su *feedback* y revisión para la versión 1.1 de este ebook.

Este libro está a la venta en <http://leanpub.com/aprendiendojavascript>, y también en [Selfy](#) y [Amazon](#).

Publicado por carlosazaustre.es

Sobre el Autor

Carlos Azaustre (Madrid, 1984) Desarrollador Web y *Technical Blogger*. Amante de JavaScript. Con varios años de experiencia tanto en empresas privadas, *Startups* y como *Freelance*. Actualmente es CTO y Co-Fundador de la *Startup* [Chefly](#)

Ingeniero en Telemática por la Universidad Carlos III de Madrid y con estudios de Máster en Tecnologías Web por la Universidad de Castilla-La Mancha (España). Fuera de la educación formal, es un amante del autoaprendizaje a través de internet. Puedes seguir sus artículos y tutoriales en su blog carlosazaustre.es

Otros libros publicados

Desarrollo Web ágil con AngularJS.



Aprende buenas prácticas en el desarrollo de aplicaciones web con el framework de JavaScript Angular.js Automatiza tus tareas cotidianas en el Frontend con el gestor GulpJS

- **Publicación:** Agosto 2014
- **Páginas:** 64
- **Lenguaje:** Ediciones en Español e Inglés
- **Autores:** Carlos Azaustre (Erica Huttner traducción en Inglés)
- [Comprar en Selfy](#)
- [Comprar en Amazon](#)

Instant Zurb Foundation 4



Get up and running in an instant with Zurb Foundation 4 Framework

- **ISBN:** 9781782164029 (Septiembre 2013)
- **Páginas:** 56
- **Lenguaje::** Inglés
- **Autores:** Jorge Arévalo y Carlos Azaustre
- [Comprar en Amazon](#)

Sobre este libro

JavaScript es el lenguaje de la web. Si necesitas programar en un navegador web, necesitas JavaScript. Bien es cierto que puedes utilizar otros lenguajes, como Dart, pero el estándar es JavaScript.

Gracias a él tenemos aplicaciones como Gmail, o Twitter, que son fuertemente dinámicas y hacen que la experiencia de uso sea mucho mejor que antaño, cuando las páginas web tenían que recargarse cada vez que realizábamos una acción.

Es un lenguaje muy demandado en la industria hoy en día, ya que además de utilizarse en el navegador, también puede usarse en el lado del servidor (Node.js). Con la multitud de frameworks que existen pueden crearse *Single Page Applications* que emulan la experiencia de una aplicación móvil en el navegador. También pueden crearse aplicaciones híbridas con herramientas como *Ionic* y *Cordova*. ¿Has oído hablar del desarrollo basado en componentes? Te sonarán entonces *Polymer* y/o *React*. Con *React Native* puedes crear aplicaciones nativas para iOS y Android con únicamente JavaScript. ¿Aplicaciones *Isomórficas*? Hoy en día todo es posible con JavaScript.

Muchas veces, si es la primera vez que te incursas en el mundo web te puede resultar abrumadora la cantidad de herramientas, preprocesadores, frameworks, etc.. Pero siempre que empezamos, cometemos el mismo error. Aprendemos la herramienta antes que el lenguaje.

Por eso me he animado a escribir este ebook que estás leyendo ahora mismo. Para enseñarte desde las bases hasta las más recientes novedades y patrones de diseño utilizando JavaScript puro (También llamado *Vanilla JS*). Una vez conoces las bases del lenguaje, ya puedes adentrarte en cualquier herramienta del mundo web.

Recientemente fue aprobado el estándar ECMAScript 6, la nueva versión de JavaScript (Actualmente utilizábamos la versión ECMAScript 5.1) que trae muchas novedades. En este ebook no he querido dejarlo de lado y hablo de cual es el equivalente en código entre la versión anterior y la presente/futura.

Espero que disfrutes del ebook tanto como yo lo he hecho escribiéndolo para ti y te sirva para tu carrera profesional. Cualquier cosa que necesites me encuentras en mi blog:

- [Carlos Azaustre Blog | Web Developer - Technical Blogger](#)

Y en las redes sociales:

- [Sígueme en Twitter @carlosazaustre](#)
- [Sígueme en Facebook](#)
- [Estoy en Instragram @carlosazaustre](#)
- [También en Google+ con +CarlosAzaustre](#)
- [Snapea conmigo en Snapchat: cazaustre](#)

Sin más, te dejo con el ebook. ¡Disfruta y aprende!

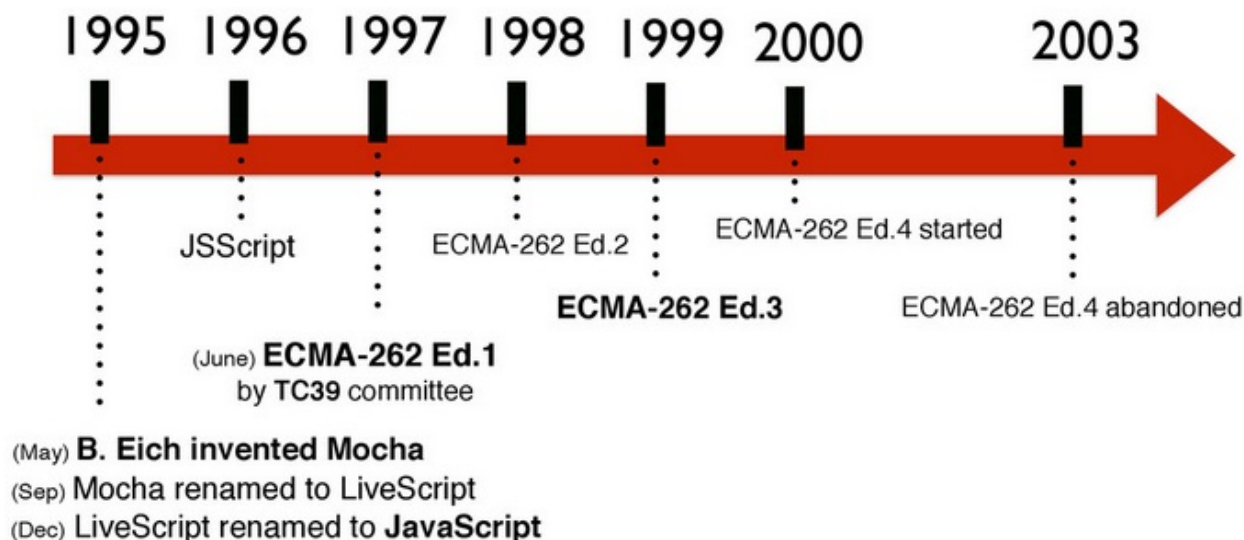
Breve historia de JavaScript

Antes de empezar con las particularidades del lenguaje, es conveniente conocer un poco de historia. De dónde viene JavaScript y cómo ha crecido su popularidad en los últimos años. ¡Prometo ser rápido y pasar cuanto antes al código!

Orígenes

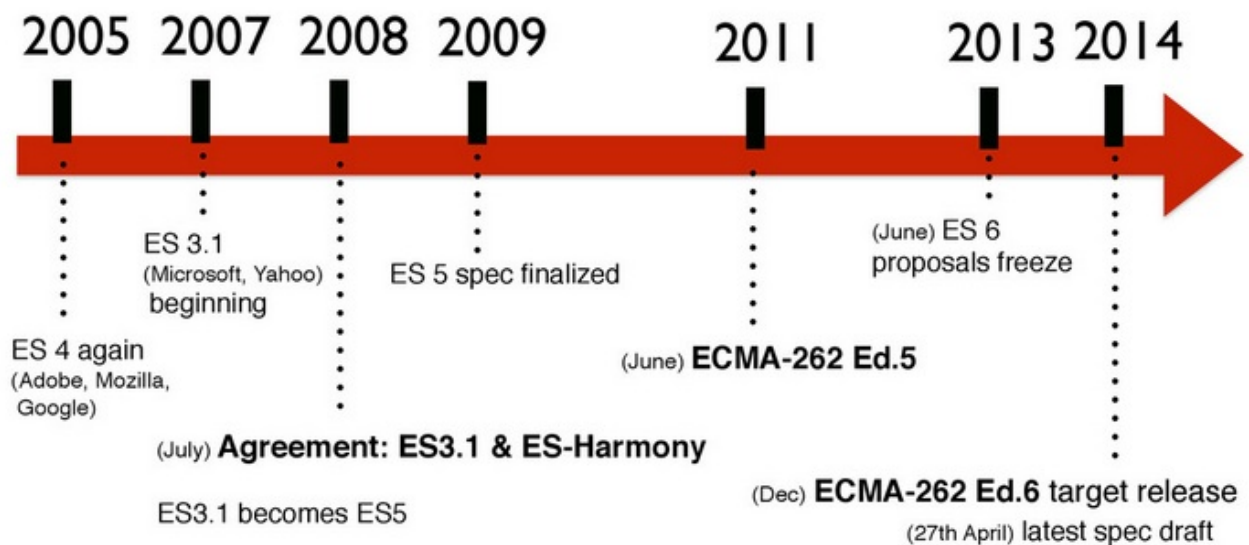
En 1995, Brendan Eich (ex-CEO de Mozilla) desarrolló lo que sería la primera versión de JavaScript para el navegador Netscape Navigator. En aquel momento se llamó *Mocha* y después fue renombrado a *LiveScript*. El nombre de *JavaScript* se le dió debido a que Netscape añadió compatibilidad con *Java* en su navegador y era una tecnología muy popular en aquel momento. Además Netscape fue adquirida por Sun Microsystems, propietaria de la marca *Java*. Esto supone que hoy en día haya una pequeña confusión y mucha gente confunda *Java* con *JavaScript* o lo considere una extensión del lenguaje, pero no es cierto, hay que aclarar que *Java* y *JavaScript* no tienen nada que ver.

En 1997 se crea un comité (llamado TC39) para crear un estándar de JavaScript por la *European Computer Manufacturers Association*, *ECMA*. En ese comité se diseña el estándar del DOM, *Document Object Model* para, de esta manera, evitar incompatibilidades entre los navegadores. Es a partir de entonces cuando los estándares de JavaScript se rigen por *ECMAScript*.



En 1999 se estandariza la versión 3 de JavaScript que se mantuvo vigente hasta hace relativamente poco. Hubo algunos intentos de lanzar una versión 4, pero la que finalmente se estandarizó y sigue hasta el momento es la versión 5 de ECMAScript, aprobada en 2011.

En Junio de 2013 el borrador de la versión 6 se quedó parado, pero en diciembre de 2014 finalmente fue aprobado y se estandarizó en Julio de 2015.

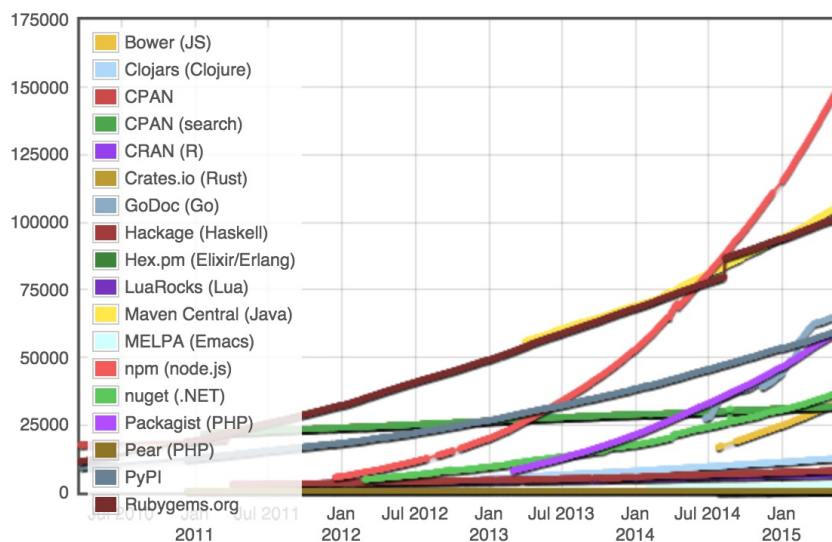


JavaScript fue diseñado para añadir efectos y animaciones a los sitios web, pero ha ido evolucionando mucho lo largo de los años, convirtiéndose en un lenguaje multipropósito. Es a partir de 2005, con la llegada de Gmail y su uso de la tecnología AJAX, *Asynchronous JavaScript And XML* (gracias al objeto *XMLHttpRequest* creado por Microsoft para Internet Explorer 5.0), lo que lanzó su popularidad.

Node.js

En 2009, Ryan Dahl creó Node.js. Node es un entorno de ejecución para JavaScript en el servidor a partir del motor V8 de renderizado de JavaScript que utiliza el navegador Chrome de Google. Node facilita la creación de aplicaciones de servidor altamente escalables. Hoy en día es muy popular para el desarrollo de Microservicios, APIs, aplicaciones web *Full-stack*, isomórficas, etc... Su comunidad es muy grande, y su sistema de paquetes y librerías NPM, *Node Package Manager*, (Aunque hoy en día ya no solo engloba paquetes de Node, también para JavaScript del lado cliente) ha superado los 150.000 módulos, convirtiéndolo en el más grande de todos por delante de Java, Ruby, PHP, etc...

Module Counts



Include

- ☒ Bower (JS)
- ☒ Clojars (Clojure)
- ☒ CPAN
- ☒ CPAN (search)
- ☒ CRAN (R)
- ☒ Crates.io (Rust)
- ☒ GoDoc (Go)
- ☒ Hackage (Haskell)
- ☒ Hex.pm (Elixir/Erlang)
- ☒ LuaRocks (Lua)
- ☒ Maven Central (Java)
- ☒ MELPA (Emacs)
- ☒ npm (node.js)
- ☒ nuget (.NET)
- ☒ Packagist (PHP)
- ☒ Pear (PHP)
- ☒ PyPI
- ☒ Rubygems.org

time period ☒ all time ☐ last year ☐ last 90 days ☐ last 30 days ☐ last 7 days

Fuente: [Module Counts](#)

Hoy en día JavaScript se utiliza en muchos sitios, Frontend, Backend, aplicaciones isomórficas, microcontroladores, *Internet of Things*, *wearables*, etc... Convirtiéndole en el lenguaje de programación del presente.

Recientemente (22 de Marzo de 2016), la web *Stackoverflow* publicó un informe a partir de una encuesta realizada a los desarrolladores usuarios de su plataforma donde resultó que **JavaScript es el lenguaje más utilizado en el mundo** solo por desarrolladores de *Frontend* si no también de *Backend*.

Toda la documentación y referencia sobre JavaScript se puede encontrar en el sitio web de desarrolladores de Mozilla [Link](#), muy recomendable de visitar cuando se tienen dudas sobre como se usa o implementa una función u objeto determinado.

Tipos de variables

JavaScript es un lenguaje débilmente tipado. Esto quiere decir que no indicamos de que tipo es cada variable que declaramos. Todas las variables admiten todos los tipos, y pueden ser reescritas. Es una de las cosas buenas y malas que tiene JavaScript.

Definición

Las variables son espacios de memoria donde almacenamos temporalmente datos desde los que podemos acceder en cualquier momento de la ejecución de nuestros programas. Tienen varios tipos y clases que veremos a continuación.

Para definir una variable en JavaScript, utilizamos la palabra reservada `var` y le damos un nombre, por ejemplo:

```
var miDato;
```

También le podemos asignar un valor en la misma línea que la declaramos, por ejemplo, a continuación a la variable `dato` le asignamos el valor `5`:

```
var dato = 5;
```

O podemos primero declarar la variable y más adelante, en otra línea, asignarle un valor:

```
var dato;  
dato = 5;
```

Debemos intentar que los nombres de las variables sean lo más descriptivos posibles, de manera que con solo leerlos sepamos que contienen y así nuestros desarrollos serán más ágiles.

Los nombres de las variables siempre han de comenzar por una letra, el símbolo `$` o `_`, nunca pueden comenzar por números u otros caracteres especiales. JavaScript también distingue entre mayúsculas o minúsculas, por tanto no es lo mismo `miDato`

que `miDato` o `miDato` , para JavaScript son nombres diferentes y las tratará de manera diferente.

Tipos

JavaScript tiene 4 tipos primitivos de datos para almacenar en variables. Estos son:

- `number`
- `boolean`
- `string`
- `undefined`

number

Sirve para almacenar **valores numéricos**. Son utilizados para contar, hacer cálculos y comparaciones. Estos son algunos ejemplos:

```
var miEntero = 1;  
var miDecimal = 1.33;
```

boolean

Este tipo de dato **almacena un bit que indica** `true` o `false` . Los valores **booleanos** se utilizan para indicar estados. Por ejemplo, asignamos a una variable el estado `false` al inicio de una operación, y al finalizarla lo cambiamos a `true` . Después realizamos la comprobación necesaria.

```
var si = true;  
var no = false;
```

string

Las variables de tipo *string* **almacenan caracteres o palabras**. Se delimitan entre comillas simples o dobles. Ejemplo:

```
var dato = "Esto es un string";  
var otroDato = 'Esto es otro string';
```

undefined

Este tipo se utiliza cuando el valor de una variable no ha sido definido aún o no existe.

Por ejemplo:

```
var dato; // su valor es undefined  
var dato = undefined;
```

Otro tipo de almacenamiento de datos que tiene JavaScript son los Objetos. En JavaScript todo es un objeto, hasta las funciones. Todo *hereda* de la clase `Object`. Se pueden definir como una estructura donde se agregan valores. Dentro de las clases que heredan de `Object` tenemos `Array`, `Date`, etc... que veremos más adelante.

Operadores

Operadores aritméticos

JavaScript posee operadores para tipos y objetos. Estos operadores permiten formar expresiones. Las más comunes son las operaciones aritméticas.

- **Suma de números:** `5 + 2`
- **Resta:** `5 - 2`
- **Operaciones con paréntesis:** `(3 + 2) - 5`
- **Divisiones:** `3 / 3`
- **Multiplicaciones:** `6 * 3`

El operador suma `+` también puede usarse para concatenar *strings* de la siguiente manera: `"Hola " + "mundo" + "!"` tendrá como resultado `"Hola mundo!"`.

JavaScript también posee los operadores post y pre incremento y decremento que añaden uno o restan uno a la variable numérica en la que se aplican. Dependiendo si son pre o post, la variable es autoincrementada o decrementada antes o después de la sentencia. Veamos un ejemplo:

```
var x = 1;      // x=1
var y = ++x;    // x=2, y=2
var z = y++ + x; // x=2, y=3, z=4
```

Como vemos en el ejemplo, la sentencia `y = ++x` lo que hace es incrementar `x`, que valía 1 y pasa a tener el valor 2, y la asignación `y = ++x` hace que `y` valga lo que `x`, es decir 2.

En la última sentencia tenemos un **postincremento**, esto lo que hace es primero evaluar la expresión y después realizar el incremento. Es decir en el momento de la asignación `z = y++ + x`, `y` vale 2 y `x` también 2, por lo tanto `z` vale 4, y después de esta asignación `y` es incrementada pasando a tener el valor 3.

Operador `typeof`

El operador `typeof` es un operador especial que nos permite conocer el tipo que tiene la variable sobre la que operamos. Ejemplos:

```
typeof 5;           // number
typeof false;       // boolean
typeof "Carlos";    // string
typeof undefined;   // undefined
```

Esto es muy útil para conocer en un momento dado que tipo estamos utilizando y prevenir errores en el desarrollo.

Operadores booleanos

Los tipos booleanos solo tienen dos valores posibles: `true` y `false` (Verdadero y Falso). Pero disponen de varios operadores que nos permiten transformar su valor.

Negación

Este operador convierte un valor booleano en su opuesto. Se representa con el signo `!`. Si se utiliza dos veces, nos devuelve el valor original.

```
!true = false
!false = true
!!true = true
```

Identidad o Igualdad

El operador de igualdad (o igualdad débil), se representa con `==` y el de identidad (o igualdad estricta), con `===`. Se recomienda el uso del operador de identidad (los 3 iguales) frente al de igualdad débil ya que el coste de procesamiento de éste último es mucho mayor y sus resultados en ocasiones son impredecibles. Es una de las *partes malas* de JavaScript, pero si se tiene cuidado no tiene porqué darnos ningún problema.

La desigualdad estricta se representa con `!==`.

```
true === true    // true
true === false   // false
true !== false   // true
true !== true    // false
```


Comparación

Podemos comparar si dos valores son menores, mayores o iguales con los operadores de comparación representados por los símbolos `<`, `>`, `<=` y `>=`. El resultado de la comparación nos devuelve `true` o `false` dependiendo de si es correcto o no.

```
5 > 3    // true
5 < 3    // false
3 >= 3   // true
2 <= 1   // false
"a" < "b" // true
```

Aunque se pueden utilizar comparaciones entre booleanos, *strings* y objetos se recomienda no usarlos ya que el orden que siguen no es muy intuitivo.

Operadores lógicos

Operador AND

Es un operador lógico que devuelve `true` siempre que todos los valores comparados sean `true`. Si uno de ellos es `false`, devuelve `false`. Se representa con el símbolo `&&`. Veamos un ejemplo

```
true && true    // true
true && false   // false
false && true   // false
false && false  // false
```

Es muy utilizado para devolver valores sin que estos sean modificados, por ejemplo para comprobar si una propiedad existe, etc. La lógica que sigue es: Si el primer valor es `false` devuelve ese valor, si no, devuelve el segundo:

```
0 && true
// 0, porque el número 0 se considera
// un valor "false"
1 && "Hola"
// "Hola", porque el número 1 (o distinto de 0)
// se considera un valor "true"
```

En el ejemplo comparamos 0 y `true`, como 0 es un valor que retorna `false`, nos devuelve el segundo valor que es `true`. En el segundo ejemplo 1 es un valor que retorna `true`, por lo que nos devolverá el segundo `"Hola"`.

Valores que devuelven `false`.

Hay ciertos valores en JavaScript que son evaluados como `false` y son: el número `0`, un *string* vacío `""`, el valor `null`, el valor `undefined` y `NaN`.

Operador OR

Es otro operador lógico que funciona a la inversa que AND. Devuelve `false` si los valores comparados son `false`. En el caso de que un valor sea `true` devolverá `true`. Se representa con el símbolo `||`.

```
true || true    // true
true || false   // true
false || true   // true
false || false  // false
```

También es muy utilizado para asignar valores por defecto en nuestras funciones. La lógica que sigue es: Si el primer valor es true, devuelve ese valor. Por ejemplo:

```
var port = process.env.PORT || 5000;
```

En este ejemplo, la variable `port` contendrá el valor de `process.env.PORT` siempre que esa variable esté definida, si no su valor será 5000.

Condicionales

Los condicionales son expresiones que nos permiten ejecutar una secuencia de instrucciones u otra diferente dependiendo de lo que estemos comprobando. Permiten establecer el flujo de ejecución de los programas de acuerdo a determinados estados.

Asignación condicional

Este tipo de asignación es también conocido como el *If simplificado* u *operador ternario*. un tipo de condicional que veremos más adelante. Sirve para asignar en una sola línea un valor determinado si la condición que se evalúa es `true` u otro si es `false`. La sintaxis es la siguiente:

```
condición ? valor_si_true : valor_si_false
```

Si la condición devuelve `true`, retornará el valor de `valor_si_true`, y si es `false` el valor devuelto será el de `valor_si_false`. Veamos unos ejemplos:

```
(true) 5 : 2; // Devuelve 5  
(false) 5 : 2; // Devuelve 2
```

Sentencia IF

Como hemos visto antes, dependiendo del resultado de una condición, obtenemos un valor u otro. Si el resultado de la condición requiere más pasos, en lugar de utilizar la asignación condicional, es mejor emplear la sentencia `if`. Tenemos 3 formas de aplicarlo:

if simple

```
if (condicion) {  
    bloque_de_codigo  
}
```

Si se cumple la condición dentro del paréntesis, se ejecuta el bloque de código incluido entre las llaves `{ ... }`

if/else

```
if (condicion) {  
    bloque_de_codigo_1  
}  
else {  
    bloque_de_codigo_2  
}
```

Con este tipo de sentencia, si se cumple la condición pasa como el anterior modelo, se ejecuta el bloque de código 1, y si la condición a evaluar no se cumple, se ejecuta el bloque de código 2.

if/else if

Y por último si queremos realizar varias comprobaciones, podemos concatenar varias sentencias if, else if, etc.. y se irán comprobando en orden:

```
if (condicion_1) {  
    bloque_1  
}  
else if (condicion_2) {  
    bloque_2  
}  
else if (condicion_3) {  
    bloque_3  
}  
else {  
    bloque_4  
}
```

En el ejemplo anterior, se comprueba la condición 1, si se cumple se ejecuta el bloque 1 y si no, se comprueba si cumple la condición 2 y en ese caso se ejecutaría el bloque 2, y así sucesivamente hasta que encuentre una condición que se cumpla o se ejecute el bloque 4 del último `else`.

Sentencia Switch

Con Switch, podemos sustituir un conjunto de sentencias `if-else` de una manera más legible. Se comprueba la condición, y según el caso que devuelva, ejecutará un bloque u otro. Para poder separar los bloques, se utiliza la palabra `break` que permite salir de toda la sentencia. Tiene un bloque `default` que se ejecuta en el caso de que no se cumpla ningún caso. Veamos un ejemplo, esto sería un `switch` siguiendo el ejemplo anterior del `if-else` :

```
switch(condicion) {  
  case condicion_1:  
    bloque_1  
    break;  
  case condicion_2:  
    bloque_2  
    break;  
  case condicion_3:  
    bloque_3  
    break;  
  default:  
    bloque_4  
}
```

El bloque `default` no es obligatorio.

Clases Core y Módulos de JavaScript

Además de los tipos primitivos que vimos al principio de este libro, JavaScript tiene unas clases, llamadas **Core** que forman parte del lenguaje. Las que más se utilizan son **Object**, **Number**, **Array** y **String**. Todas ellas heredan de **Object**.

Object

Un objeto es una colección de variables y funciones agrupadas de manera estructural. A las variables definidas dentro de un objeto se las denomina propiedades, y las funciones, métodos. Veamos un ejemplo de objeto que recoge los datos de un libro:

```
var libroAngular = {  
  titulo: 'Desarrollo web ágil con AngularJS',  
  autor: 'Carlos Azaustre',  
  paginas: 64,  
  formatos: ["PDF", "ePub", "Mobi"],  
  precio: 2.79,  
  publicado: false  
};
```

Como podemos ver, las propiedades son pares *clave-valor*, separados por comas, y podemos acceder a ellas de forma independiente de varias formas, con la notación punto o con la notación array:

```
libroAngular.titulo; // Desarrollo web ágil con AngularJS  
libroAngular['paginas']; // 64
```

También podemos modificarlas de la misma manera:

```
libroAngular.precio = 1.95;  
libroAngular['publicado'] = true;
```

Con la notación array, podemos acceder a las propiedades con variables. Ejemplo:

```
var propiedad = "autor";  
libroAngular[propiedad]; // "Carlos Azaustre"
```

Pero no funciona con la notación punto:

```
var propiedad = "autor";  
libroAngular.propiedad; // undefined
```

Como hemos dicho antes, si el objeto contiene funciones se les llama métodos. En el siguiente capítulo veremos como se inicializan e invocan funciones más en detalle. Si queremos crearlas dentro de un objeto e invocarlas, sería así:

```
var libroAngular = {  
  paginas: 64,  
  leer: function () {  
    console.log("He leído el libro de AngularJS");  
  }  
};  
  
libroAngular.leer(); // Devuelve: "He leído el libro de AngularJS"
```

Para crear un objeto podemos hacerlo con la notación de llaves {...} o creando una nueva instancia de clase:

```
var miObjeto = { propiedad: "valor" };  
var miObjeto = new Object({ propiedad: "valor" });
```

Anidación

Un objeto puede tener propiedades y estas propiedades tener en su interior más propiedades. Sería una representación en forma de árbol y podemos acceder a sus propiedades de la siguiente manera:

```
var libro = {
  titulo: "Desarrollo Web ágil con AngularJS",
  autor: {
    nombre: "Carlos Azaustre",
    nacionalidad: "Española",
    edad: 30,
    contacto: {
      email: "carlosazaustre@gmail.com",
      twitter: "@carlosazaustre"
    }
  },
  editorial: {
    nombre: "carlosazaustre.es Books",
    web: "https://carlosazaustre.es"
  }
};
// Podemos acceder con notación punto, array, o mixto.
libro.autor.nombre; // "Carlos Azaustre"
libro['autor']['edad']; // 30
libro['editorial'].web; // "https://carlosazaustre.es"
libro.autor['contacto'].twitter; // "@carlosazaustre"
```

Igualdad entre objetos

Para que dos objetos sean iguales al compararlos, deben tener la misma referencia. Debemos para ello utilizar el operador identidad `===`. Si creamos dos objetos con el mismo contenido, no serán iguales a menos que compartan la referencia. Veamos un ejemplo:

```
var coche1 = { marca: "Ford", modelo: "Focus" };
var coche2 = { marca: "Ford", modelo: "Focus"};
coche1 === coche2; // Devuelve false, no comparten referencia
coche1.modelo === coche2.modelo; // Devuelve true porque el valor es el mismo.
var coche3 = coche1;
coche1 === coche3; // Devuelve true, comparten referencia
```

Number

Es la clase del tipo primitivo `number`. Se codifican en formato de coma flotante con doble precisión (Es decir, con 64 bits / 8 bytes) y podemos representar números enteros, decimales, hexadecimales, y en coma flotante. Veamos unos ejemplos:


```
// Número entero, 25
25
// Número entero, 25.5. Los decimales se separan de la parte entera con punto `.`
25.5
// Número hexadecimal, se representa con 0x seguido del número hexadecimal
0x1F // 31 decimal
0xFF // 255 decimal
0x7DE // 2014
// Coma flotante, separamos la mantisa del exponente con la letra `e`
5.4e2 // Representa 5.4 * 10 elevado a 2 = 540
```

La clase `Number` incluye los números `Infinity` y `-Infinity` para representar números muy grandes:

```
1/0 = Infinity
-1/0 = -Infinity
1e1000 = Infinity
-1e1000 = -Infinity
```

El rango real de números sobre el que podemos operar es $\sim 1,797 \times 10^{308}$ --- 5×10^{-324} .

También disponemos del valor `NaN` (*Not A Number*) para indicar que un determinado valor no representa un número:

```
"a"/15 = NaN
```

Para crear un número podemos hacerlo con la forma primitiva o con la clase `Number`. Por simplicidad se utiliza la forma primitiva.

```
var numero = 6;
var numero = new Number(6);
```

Funciones de Number

JavaScript tiene 2 funciones interesantes para convertir un string en su número equivalente.

`parseInt()`

Devuelve el número decimal equivalente al string que se pasa por parámetro. Si se le indica la base, lo transforma en el valor correspondiente en esa base, si no, lo devuelve en base 10 por defecto. Veamos unos ejemplos:

```
parseInt("1111");    // Devuelve 1111
parseInt("1111", 2); // Devuelve 15
parseInt("1111", 16); // Devuelve 4369
```

parseFloat()

Función similar a `parseInt()` que analiza si es un número de coma flotante y devuelve su representación decimal

```
parseFloat("5e3");    // Devuelve 5000
```

number.toFixed(x)

Devuelve un string con el valor del numero `number` redondeado al alza, con tantos decimales como se indique en el parámetro `x`.

```
var n = 2.5674;
n.toFixed(0); // Devuelve "3"
n.toFixed(2); // Devuelve "2.57"
```

number.toExponential(x)

Devuelve un string redondeando la base o mantisa a `x` decimales. Es la función complementaria a `parseFloat`

```
var n = 2.5674;
n.toExponential(2); // Devuelve "2.56e+0"
```

number.toString(base)

Devuelve un string con el número equivalente `number` en la base que se pasa por parámetro. Es la función complementaria a `parseInt`

```
(1111).toString(10); // Devuelve "1111"
(15).toString(2);    // Devuelve "1111"
(4369).toString(16); // Devuelve "1111"
```

Módulo Math

`Math` es una clase propia de JavaScript que contiene varios valores y funciones que nos permiten realizar operaciones matemáticas. Estos son los más utilizados:

```
Math.PI // Número Pi = 3.14159265...
Math.E  // Número e = 2.7182818...
Math.random() // Número aleatorio entre 0 y 1, ej: 0.45673858
Math.pow(2,6) // Potencia de 2 elevado a 6 = 64;
Math.sqrt(4)  // raíz cuadrada de 4 = 2
Math.min(4,3,1) // Devuelve el mínimo del conjunto de números = 1
Math.max(4,3,1) // Devuelve el máximo del conjunto de números = 4
Math.floor(6.4) // Devuelve la parte entera más próxima por debajo, en este caso 6
Math.ceil(6.4)  // Devuelve la parte entera más próxima por encima, en este caso 7
Math.round(6.4) // Redondea a la parte entera más próxima, en este caso 6
Math.abs(x);    // Devuelve el valor absoluto de un número

// Funciones trigonométricas
Math.sin(x);    // Función seno de un valor
Math.cos(x);    // Función coseno de un valor
Math.tan(x);    // Función tangente de un valor
Math.log(x);    // Función logaritmo
...
```

Existen muchos más, puedes consultarlo en la documentación de Mozilla: [link](#)

Array

Es una colección de datos que pueden ser números, strings, objetos, otros arrays, etc... Se puede crear de dos formas con el literal `[...]` o creando una nueva instancia de la clase `Array`

```
var miArray = [];
var miArray = new Array();
```

```
var miArray = [1, 2, 3, 4]; // Array de números
var miArray = ["Hola", "que", "tal"]; // Array de Strings
var miArray = [ {propiedad: "valor1" }, { propiedad: "valor2" }]; // Array de objetos
var miArray = [[2, 4], [3, 6]]; // Array de arrays, (Matriz);
var miArray = [1, true, [3,2], "Hola", {clave: "valor"}]; // Array mixto
```

Se puede acceder a los elementos del array a través de su índice y con `length` conocer su longitud.

```
var miArray = ["uno", "dos", "tres"];
miArray[1]; // Devuelve: "dos"
miArray.length; // Devuelve 3
```

Si accedemos a una posición que no existe en el array, nos devuelve `undefined`.

```
miArray[8]; // undefined
```

Métodos

`Array` es una clase de JavaScript, por tanto los objetos creados a partir de esta clase heredan todos los métodos de la clase padre. Los más utilizados son:

```
var miArray = [3, 6, 1, 4];
miArray.sort(); // Devuelve un nuevo array con los valores ordenados: [1, 3, 4, 6]
miArray.pop(); // Devuelve el último elemento del array y lo saca. Devuelve 6 y miArr
miArray.push(2); // Inserta un nuevo elemento en el array, devuelve la nueva longitud
miArray.reverse(); // Invierte el array, [2,4,3,1]
```

Otro método muy útil es `join()` sirve para crear un string con los elementos del array uniéndolos con el "separador" que le pasemos como parámetro a la función. Es muy usado para imprimir strings, sobre todo a la hora de crear templates. Ejemplo:

```
var valor = 3;
var template = [
  "<li>",
  valor,
  "</li>"
].join("");

console.log(template); // Devuelve: "<li>3</li>"
```

Lo cual es mucho más eficiente en términos de procesamiento, que realizar lo siguiente, sobre todo si estas uniones se realizan dentro de bucles.

```
var valor = 3;
var template = "<li>" + valor + "</li>";
```

Si queremos aplicar una misma función a todos los elementos de un array podemos utilizar el método `map`. Imaginemos el siguiente array de números `[2, 4, 6, 8]` y queremos conocer la raíz cuadrada de cada uno de los elementos podríamos hacerlo así:

```
var miArray = [2, 4, 6, 8];
var raices = miArray.map(Math.sqrt);
});

// raices: [ 1.4142135623730951, 2, 2.449489742783178, 2.8284271247461903 ]
```

O algo más específico:

```
var miArray = [2, 4, 6, 8];
var resultados = miArray.map(function(elemento) {
    return elemento * 2;
});

// resultados: [ 4, 8, 12, 16 ]
```

Otra función interesante de los arrays es la función `filter`. Nos permite "filtrar" los elementos de un array dada una condición sin necesidad de crear un bucle (que veremos más adelante) para iterarlo. Por ejemplo, dado un array con los números del 1 al 15, obtener un array con los números que son divisibles por 3:

```
var miArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];
var resultado = miArray.filter(function(elemento) {
    return elemento % 3 === 0;
});

// resultados: [ 3, 6, 9, 12, 15 ]
```

Si queremos obtener una parte del array, podemos emplear la función `slice` pasándole por parámetro el índice a partir del que queremos cortar y el final. Si no se indica el parámetro de fin, se hará el "corte" hasta el final del array, si no, se hará hasta la posición indicada y si se pasa un número negativo, contará desde el final del array hacia atrás.

El método devuelve un nuevo array sin transformar sobre el que se está invocando la función. Veamos unos ejemplos:

```
var miArray = [4, 8, 15, 16, 23, 42];
miArray.slice(2); // [15, 16, 23, 42]
miArray.slice(2, 4); // [15, 16] (la posición de fin no es inclusiva)
miArray.slice(2, -1); // [15, 16, 23]
miArray.slice(2, -2); // [15, 16]
```

String

Como vimos al principio de este libro, los strings son un tipo de variable primitivo en JavaScript pero también, al igual que con `Number` tienen su clase propia y métodos.

Un string se comporta como un *Array*, no es más que un conjunto de caracteres, con índices que van desde el 0 para el primer carácter hasta el último. Veamos algunos ejemplos de como acceder a los caracteres y los métodos que posee esta clase

```
// Supongamos el string con el texto "javascript"
"javascript"[2] // Acceso como array, devuelve el tercer carácter "v", ya que la prim
"javascript".length() // Devuelve 10
"javascript".charAt(2) // Devuelve el caracter en formato UNICODE de "v", el 118
"javascript".indexOf("script"); // Devuelve el índice donde comienza el string "scrip
"javascript".substring(4,10); // Devuelve la parte del string comprendida entre los i
```

Para crear un string podemos hacerlo con notación de tipo o creando un nuevo objeto. Por simplicidad se utiliza la forma primitiva.

```
var texto = "Hola Mundo";
var texto = new String("Hola Mundo");
```

Un string puede ser transformado en array con el método `split()` pasándole como parámetro el delimitador que queramos que separe los elementos. Por ejemplo:

```
var fecha = new Date();
fecha = fecha.toString(); // "Wed May 20 2015 20:16:25 GMT+0200 (CEST)"
fecha = fecha.split(" "); // ["Wed", "May", "20", "2015", "20:16:25", "GMT+0200", "(C
fecha[4]; // "20:16:25"
```


Funciones

Las funciones en JavaScript son bloques de código ejecutable, a los que podemos pasar parámetros y operar con ellos. Nos sirven para modular nuestros programas y estructurarlos en bloques que realicen una tarea concreta. De esta manera nuestro código es más legible y mantenible.

Las funciones normalmente, al acabar su ejecución devuelven un valor, que conseguimos con el parámetro `return`. Se declaran con la palabra reservada `function` y a continuación suelen llevar un nombre, para poder invocarlas más adelante. Si no llevan nombre se les llama funciones anónimas.

Veamos un ejemplo de función:

```
function saludar (nombre) {  
  return ("Hola " + nombre + "!");  
}  
  
saludar("Carlos"); // Devuelve "Hola Carlos!"
```

La función del ejemplo se llama `saludar`, y se le pasa un único parámetro, entre paréntesis `(...)`, que es `nombre`. Ese parámetro funciona como contenedor de una variable que es utilizada dentro del bloque de código delimitado por las llaves `{...}`. El comando `return` devolverá el String que concatena texto con el valor que contiene el parámetro `nombre`.

Si no pasásemos ningún valor por parámetro, obtendríamos el valor `undefined`.

```
function saludar (nombre) {  
  return ("Hola " + nombre + "!");  
}  
  
saludar(); // Devuelve "Hola undefined!"
```

También podemos acceder a los parámetros que se pasan por argumento a través del array `arguments` sin indicarlo en la definición de la función, aunque esta opción no es muy utilizada. Ejemplo:


```
function saludar () {  
  var tipo = arguments[0];  
  var nombre = arguments[1];  
  return (tipo + ", " + nombre + "!");  
}  
  
saludar("Adios", "Carlos"); // Devuelve "Adios, Carlos!"
```

Parámetros por defecto

Una buena práctica para evitar errores o que se tome el valor `undefined` sin que podamos controlarlo, es utilizar algunos de los operadores booleanos que vimos en capítulos anteriores. Si tomamos el operador OR `||` podemos asignar un valor por defecto si no está definido. Veamos un ejemplo:

```
function saludar (tipo, nombre) {  
  var tipo = tipo || "Hola";  
  var nombre = nombre || "Carlos";  
  return (tipo + ", " + nombre + "!");  
}  
  
saludar(); // "Hola, Carlos!"  
saludar("Adios"); // "Adios, Carlos!"  
saludar("Hasta luego", "Pepe"); // "Hasta luego, Pepe!"
```

Ámbito de una función.

Por defecto, cuando declaramos una variable con `var` la estamos declarando de forma global y es accesible desde cualquier parte de nuestra aplicación. Tenemos que tener cuidado con los nombres que elegimos ya que si declaramos a una variable con el mismo nombre en varias partes de la aplicación estaremos sobrescribiendo su valor y podemos tener errores en su funcionamiento.

Si declaramos una variable dentro de una función, esta variable tendrá un ámbito local al ámbito de esa función, es decir, solo será accesible de la función hacia adentro. Pero si la definimos fuera de una función, tendrá un ámbito global.

En la versión 6 de ECMAScript tenemos los tipos de variable `let` y `const` en lugar de `var` y definen unos ámbitos específicos. `const` crea una constante cuyo valor no cambia durante el tiempo y `let` define el ámbito de la variable al ámbito donde ha sido

definida (por ejemplo en una función).

Con un ejemplo lo veremos más claro:

```
var valor = "global";

function funcionlocal () {
  var valor = "local";
  return valor;
}

console.log(valor);           // "global"
console.log(funcionLocal()); // "local"
console.log(valor);           // "global"
```

Aunque tenemos definida fuera de la función la variable `valor`, si dentro de la función la declaramos y cambiamos su valor, no afecta a la variable de fuera porque su ámbito (o *scope*) de ejecución es diferente. Una definición de variable local tapa a una global si tienen el mismo nombre.

Closures

Los *Closures* o funciones cierre son un patrón de diseño muy utilizado en JavaScript y son una de las llamadas *Good parts*. Para poder comprender su funcionamiento veamos primero unos conceptos.

Funciones como objetos

Las funciones en JavaScript son objetos, ya que todo en JavaScript es un objeto, heredan sus propiedades de la clase `Object`. Entonces pueden ser tratadas como tal. Podemos guardar una función en una variable y posteriormente invocarla con el operador paréntesis `()`. Ejemplo:

```
var saludar = function (nombre) {
  return "Hola " + nombre;
};

saludar("Carlos"); // "Hola Carlos"
```

Si a la variable que guarda la función no la invocamos con el operador paréntesis, el resultado que nos devolverá es el código de la función

```
saludar; // Devuelve 'function(nombre) { return "Hola " + nombre };'
```

Funciones anidadas

Las funciones pueden tener otras funciones dentro de ellas, produciendo nuevos ámbitos para las variables definidas dentro de cada una. Y para acceder desde el exterior a las funciones internas, tenemos que invocarlas con el operador doble paréntesis `()()`. Veamos un ejemplo

```
var a = "OLA";

function global () {
  var b = "K";

  function local () {
    var c = "ASE";
    return a + b + c;
  }

  return local;
}

global(); // Devuelve la función local: "function local() { var c = "ASE"...""
global()(); // Devuelve la ejecución de la función local: "OLAKASE"

var closure = global();
closure(); // Devuelve lo mismo que global()(): "OLAKASE"
```

Vistos estos conceptos ya podemos definir lo que es un `closure`.

Función cierre o closure

Un *Closure* es una función que encapsula una serie de variables y definiciones locales que únicamente serán accesibles si son devueltas con el operador `return`. JavaScript al no tener una definición de clases como tal (como por ejemplo en Java, aunque con la versión ECMAScript6 esto cambia un poco) este patrón de creación de closures, hace posible modularizar nuestro código y crear algo parecido a las clases.

Veamos un ejemplo de closure con la siguiente función. Creamos una función que tiene un variable local que guarda el valor de un numero que será incrementado o decrementado según llamemos a las funciones locales que se devuelven y acceden a

esa variable. la variable local `_contador` no puede ser accesible desde fuera si no es a través de esas funciones:

```
var miContador = (function () {  
    var _contador = 0; // Por convención, a las variables "privadas" se las llama con un  
  
    function incrementar () {  
        return _contador++;  
    }  
  
    function decrementar () {  
        return _contador--;  
    }  
  
    function valor () {  
        return _contador;  
    }  
  
    return {  
        incrementar: incrementar,  
        decrementar: decrementar,  
        valor: valor  
    }  
})();  
  
miContador.valor(); // 0  
miContador.incrementar();  
miContador.incrementar();  
miContador.valor(); // 2  
miContador.decrementar();  
miContador.valor(); // 1
```

Funciones como clases

Un closure es muy similar a una clase, la principal diferencia es que una clase tendrá un constructor que cumple el mismo cometido que el closure. Al crear un objeto a partir de una clase debemos usar el parámetro `new` y si es un closure, al inicializar un nuevo objeto, se le pasa lo que le devuelve la función cierre.

Veamos un ejemplo de la misma función, codificada como clase y como closure, y como se crearían sus objetos.

```
function inventario (nombre) {  
  var _nombre = nombre;  
  var _articulos = {};  
  
  function add (nombre, cantidad) {  
    _articulos[nombre] = cantidad;  
  }  
  
  function borrar (nombre) {  
    delete _articulos[nombre];  
  }  
  
  function cantidad (nombre) {  
    return _articulos[nombre];  
  }  
  
  function nombre () {  
    return _nombre;  
  }  
  
  return {  
    add: add,  
    borrar: borrar,  
    cantidad: cantidad,  
    nombre: nombre  
  }  
}
```

Una vez construido la closure, podemos usar sus métodos como vemos a continuación:

```
var libros = inventario("libros");  
libros.add("AngularJS", 3);  
libros.add("JavaScript", 10);  
libros.add("NodeJS", 5);  
libros.cantidad("AngularJS"); // 3  
libros.cantidad("JavaScript"); // 10  
libros.borrar("JavaScript");  
libros.cantidad("JavaScript"); // undefined
```

Ahora veamos como sería esto mismo pero codificado como Clase:

```
function Inventario (nombre) {  
  this.nombre = nombre;  
  this.articulos = [];  
  
  this.add = function (nombre, cantidad) {  
    this.articulos[nombre] = cantidad;  
  }  
  
  this.borrar = function (nombre) {  
    delete this.articulos[nombre];  
  }  
  
  this.cantidad = function (nombre) {  
    return this.articulos[nombre];  
  }  
  
  this.getNombre = function () {  
    return this.nombre;  
  }  
}
```

Una vez definida la clase, crear objetos a partir de ella e invocar a sus métodos sería así:

```
var libros = new Inventario("Libros");  
libros.add("AngularJS", 3);  
libros.add("JavaScript", 10);  
libros.add("NodeJS", 5);  
libros.cantidad("AngularJS"); // 3  
libros.cantidad("JavaScript"); // 10  
libros.borrar("JavaScript");  
libros.cantidad("JavaScript"); // undefined
```

Esta forma de codificar las funciones como clases se conoce como *Factory Pattern* o *Template functions*.

Uso de Prototype

Un problema importante que tiene este tipo de estructura, es que cuando creamos un nuevo objeto a partir de esta clase, reservará espacio en memoria para toda la clase incluyendo atributos y métodos. Con un objeto solo creado no supone mucha desventaja, pero imaginemos que creamos varios objetos:

```
var libros = new Inventario("Libros");  
var discos = new Inventario("discos");  
var juegos = new Inventario("juegos");  
var comics = new Inventario("comics");  
...
```

Esto supone que las funciones de la clase, `add`, `borrar`, `cantidad` y `getNombre` están siendo replicadas en memoria, lo que hace que sea ineficiente.

```
> libros
< ▼ Inventario {nombre: "libros", articulos: Array[0]} ⓘ
  ▶ add: function (nombre, cantidad)
  ▶ articulos: Array[0]
  ▶ borrar: function (nombre)
  ▶ cantidad: function (nombre)
  ▶ getNombre: function ()
    nombre: "libros"
  ▶ __proto__: Inventario

> discos
< ▼ Inventario {nombre: "discos", articulos: Array[0]} ⓘ
  ▶ add: function (nombre, cantidad)
  ▶ articulos: Array[0]
  ▶ borrar: function (nombre)
  ▶ cantidad: function (nombre)
  ▶ getNombre: function ()
    nombre: "discos"
  ▶ __proto__: Inventario

> comics
< ▼ Inventario {nombre: "comics", articulos: Array[0]} ⓘ
  ▶ add: function (nombre, cantidad)
  ▶ articulos: Array[0]
  ▶ borrar: function (nombre)
  ▶ cantidad: function (nombre)
  ▶ getNombre: function ()
    nombre: "comics"
  ▶ __proto__: Inventario

> juegos
< ▼ Inventario {nombre: "juegos", articulos: Array[0]} ⓘ
  ▶ add: function (nombre, cantidad)
  ▶ articulos: Array[0]
  ▶ borrar: function (nombre)
  ▶ cantidad: function (nombre)
  ▶ getNombre: function ()
    nombre: "juegos"
  ▶ __proto__: Inventario

>
```

Para solucionar esto podemos hacer uso del objeto `Prototype` que permite que objetos de la misma clase compartan métodos y no sean replicados en memoria de manera ineficiente. La forma correcta de implementar la clase `Inventario` sería la siguiente:


```
function Inventario (nombre) {  
  this.nombre = nombre;  
  this.articulos = [];  
};  
  
Inventario.prototype = {  
  add: function (nombre, cantidad) {  
    this.articulos[nombre] = cantidad;  
  },  
  
  borrar: function (nombre) {  
    delete this.articulos[nombre];  
  },  
  
  cantidad: function (nombre) {  
    return this.articulos[nombre];  
  },  
  
  getNombre: function () {  
    return this.nombre;  
  }  
};
```

De esta manera, si queremos crear un nuevo objeto de la clase `Inventario` y usar sus métodos, lo podemos hacer como veníamos haciendo hasta ahora, sólo que internamente será más eficiente el uso de la memoria por parte de JavaScript y obtendremos una mejora en el rendimiento de nuestras aplicaciones.

Creando de nuevo los objetos `libros`, `discos`, `juegos` y `comics`, su espacio en memoria es menor (Ya no tienen replicados los métodos):

```

> libros
< ▼ Inventario {nombre: "Libros", articulos: Array[0]} ⓘ
  ► articulos: Array[0]
  nombre: "Libros"
  ► __proto__: Object

> discos
< ▼ Inventario {nombre: "discos", articulos: Array[0]} ⓘ
  ► articulos: Array[0]
  nombre: "discos"
  ► __proto__: Object

> comics
< ▼ Inventario {nombre: "comics", articulos: Array[0]} ⓘ
  ► articulos: Array[0]
  nombre: "comics"
  ► __proto__: Object

> juegos
< ▼ Inventario {nombre: "juegos", articulos: Array[0]} ⓘ
  ► articulos: Array[0]
  nombre: "juegos"
  ► __proto__: Object

>

```

```

var libros = new Inventario('libros');
libros.getNombre();
libros.add("AngularJS", 3);
...
var comics = new Inventario('comics');
comics.add("The Walking Dead", 10);
...

```

Clases en ECMAScript 6

Con la llegada de la nueva versión del estándar de JavaScript (ECMAScript 6 o ECMAScript 2015) la definición de una función como clase ha cambiado. ES6 aporta un *azúcar sintáctico* para declarar una clase como en la mayoría de los lenguajes de programación orientados a objetos, pero por *debajo* sigue siendo una función prototipal.

El ejemplo anterior del `Inventario`, transformado a ES6 sería tal que así

```
class Inventario {  
  constructor(nombre) {  
    this.nombre = nombre;  
    this.articulos = [];  
  }  
  
  add (nombre, cantidad) {  
    this.articulos[nombre] = cantidad;  
  }  
  
  borrar (nombre) {  
    delete this.articulos[nombre]  
  }  
  
  cantidad (nombre) {  
    return this.articulos[nombre]  
  }  
  
  getNombre () {  
    return this.nombre;  
  }  
}
```

Utilizando la palabra reservada `class` creamos una clase que sustituye a la función prototipal de la versión anterior.

El método especial `constructor` sería el que se definía en la función constructora anterior. Después los métodos `add`, `borrar`, `cantidad` y `getNombre` estarían dentro de la clase y sustituirían a las funciones prototipales de la versión ES5.

Su utilización es igual que en la versión anterior

```
var libros = new Inventario("Libros");  
  
libros.add("AngularJS", 3);  
libros.add("JavaScript", 10);  
libros.add("NodeJS", 5);  
  
libros.cantidad("AngularJS"); // 3  
libros.cantidad("JavaScript"); // 10  
libros.borrar("JavaScript");  
libros.cantidad("JavaScript"); // undefined
```

Con esta nueva sintaxis podemos implementar herencia de una forma muy sencilla. Imagina que tienes una clase `Vehículo` de la siguiente manera:

```
class Vehiculo {
  constructor (tipo, nombre, ruedas) {
    this.tipo = tipo;
    this.nombre = nombre;
    this.ruedas = ruedas
  }

  getRuedas () {
    return this.ruedas
  }

  arrancar () {
    console.log(`Arrancando el ${this.nombre}`)
  }

  aparcacar () {
    console.log(`Aparcando el ${this.nombre}`)
  }
}
```

Y quieres crear ahora una clase `Coche` que herede de vehículo para poder utilizar los métodos que esta tiene. Esto lo podemos hacer con la clase reservada `extends`

y con `super()` llamamos al constructor de la clase que hereda

```
class Coche extends Vehiculo {
  constructor (nombre) {
    super('coche', nombre, 4)
  }
}
```

Si ahora creamos un nuevo objeto `Coche` podemos utilizar los métodos de la clase `Vehiculo`

```
let fordFocus = new Coche('Ford Focus')
fordFocus.getRuedas() // 4
fordFocus.arrancar() // Arrancando el Ford Focus
```

Bucles

En ocasiones nos interesa que determinados bloques de código se ejecuten varias veces mientras se cumpla una condición. En ese caso tenemos los bucles para ayudarnos. Dependiendo de lo que necesitemos usaremos uno u otro. A continuación veremos cuales son y algunos ejemplos prácticos para conocer su uso.

Existen 3 elementos que controlan el flujo del bucle. La **inicialización** que fija los valores con los que iniciamos el bucle. La condición de **permanencia** en el bucle y la **actualización** de las variables de control al ejecutarse la iteración.

Bucle while

La sintaxis de un bucle `while` es la siguiente, el bloque de código dentro del `while` se ejecutará mientras se cumpla la condición.

```
var condicion; // Inicialización

while (condicion) { // Condición de permanencia
  bloque_de_codigo // Código a ejecutar y actualización de la variable de control
}
```

Por ejemplo si queremos mostrar por consola los números del 1 al 10, con un bucle `while` sería así:

```
var i = 1; // Inicialización
while (i < 11) { // Condición de permanencia
  console.log(i); // Código a ejecutar
  i++; // Actualización de la variable de control
}

// Devuelve: 1 2 3 4 5 6 7 8 9 10
```

Bucle Do/While

El bucle `do-while` es similar al `while` con la salvedad de que ejecutamos un bloque de código dentro de `do` por primera vez y después se comprueba la condición de permanencia en el bucle. De esta manera nos aseguramos que al menos una vez el bloque se ejecute

```
var i = 1;
do {
  console.log(i);
  i++;
} while (i < 11);

// Devuelve: 1 2 3 4 5 6 7 8 9 10
```

Bucle For

Por último el bucle `for` es una sentencia especial pero muy utilizada y potente. Nos permite resumir en una línea la forma de un bucle `while`. Su sintaxis es la siguiente:

```
for(inicializacion; condición de permanencia; actualizacion) {
  bloque_de_codigo
}
```

Los elementos de control se definen entre los paréntesis `(...)` y se separan por punto y coma `;`. Los bucles anteriores en formato `for` serían así:

```
for (var i=1; i < 11; i++) {
  console.log(i);
}

// Devuelve: 1 2 3 4 5 6 7 8 9 10
```

Buenas prácticas en bucles For.

Es común que en nuestros desarrollos utilicemos éste tipo de bucle a menudo, por ejemplo para recorrer arrays. Si tomamos unas consideraciones en cuenta, podemos hacer programas más eficientes.

Por ejemplo, un bucle de este tipo:

```
var objeto = {  
  unArray: new Array(10000);  
};  
  
for(var i=0; i<objeto.unArray.length; i++) {  
  objeto.unArray[i] = "Hola!";  
}
```

Tiene varios puntos, donde perdemos rendimiento. El primero de ellos es comprobar la longitud del array dentro de la definición del bucle `for`. Esto hace que en cada iteración estemos comprobando la longitud y son pasos que podemos ahorrarnos y que harán más eficiente la ejecución. Lo ideal es *cachear* este valor en una variable, ya que no va a cambiar.

```
var longitud = objeto.unArray.length;
```

Y esta variable la podemos incluir en el bucle `for` en la inicialización, separada por una coma `,`, quedando de la siguiente manera:

```
for (var i=0, longitud=objeto.unArray.length; i<longitud; i++) {  
  ...  
}
```

Otra optimización que podemos hacer es *cachear* también el acceso al array dentro del objeto `grandesCitas`. De ésta manera nos ahorramos un paso en cada iteración al acceder al interior del bucle. A la larga son varios milisegundos que salvamos y afecta al rendimiento:

```
var unArray = objeto.unArray;  
for (var i=0, longitud=unArray.length; i<longitud; i++) {  
  unArray[i] = "Hola!";  
}
```

Si utilizamos los comandos `console.time` y `console.timeEnd` podemos ver cuanto tiempo llevó la ejecución:

```
> console.time('Test');  
for(var i=0; i<objeto.unArray.length; i++) {  
    objeto.unArray[i] = "Hola";  
}  
console.timeEnd('Test');
```

Test: 23.081ms

< undefined

```
> console.time('Test');  
var unArray = objeto.unArray;  
for(var i=0, longitud=unArray.length; i<longitud; i++) {  
    unArray[i] = "Hola";  
}  
console.timeEnd('Test');
```

Test: 17.437ms

Como se puede comprobar, es más rápido de la segunda manera.

Bucle For/Each

Este tipo de bucle fue una novedad que introdujo ECMAScript5. Pertenece a las funciones de la clase `Array` y nos permite iterar dentro de un array de una manera secuencial. Veamos un ejemplo:

```
var miArray = [1, 2, 3, 4];  
miArray.forEach(function (elemento, index) {  
    console.log("El valor de la posición " + index + " es: " + elemento);  
});  
  
// Devuelve lo siguiente  
// El valor de la posición 0 es: 1  
// El valor de la posición 1 es: 2  
// El valor de la posición 2 es: 3  
// El valor de la posición 3 es: 4
```

¿Quieres ver una utilidad del `forEach`? Imagina que quieres recorrer los valores y propiedades de un objeto de este tipo:


```
var libro = {  
  titulo: "Aprendiendo JavaScript",  
  autor: "Carlos Azaustre",  
  numPaginas: 64,  
  editorial: "carlosazaustre.es",  
  precio: "2.95"  
};
```

Con `forEach` no puedes, porque es un método de la clase `Array`, por tanto necesitas que las propiedades del objeto sean un array. Esto lo puedes conseguir haciendo uso de las funciones de la clase `Object`: `getOwnPropertyNames` que devuelve un array con todas las propiedades del objeto y con `getOwnPropertyDescriptor` accedes al valor. Veamos un ejemplo:

```
var propiedades = Object.getOwnPropertyNames(libro);  
propiedades.forEach(function(name) {  
  var valor = Object.getOwnPropertyDescriptor(libro, name).value;  
  console.log("La propiedad " +name+ " contiene: " +valor );  
});  
  
// Devuelve:  
// La propiedad titulo contiene: Aprendiendo JavaScript  
// La propiedad autor contiene: Carlos Azaustre  
// La propiedad numPaginas contiene: 64  
// La propiedad editorial contiene: carlosazaustre.es  
// La propiedad precio contiene: 2.95
```

Bucle For/In

Además de `ForEach`, tenemos el bucle `ForIn`. Con este tipo de bucle podemos iterar entre las propiedades de un objeto de una manera más sencilla que la vista anteriormente. La sintaxis es `for(key in object)` siendo `key` el nombre de la propiedad y `object[key]` el valor de la propiedad. Como siempre, veamos un ejemplo práctico:

```
var libro = {  
  titulo: "Aprendiendo JavaScript",  
  autor: "Carlos Azaustre",  
  numPaginas: 64,  
  editorial: "carlosazaustre.es",  
  precio: "2.95"  
};  
  
for(var prop in libro) {  
  console.log("La propiedad " +prop+ " contiene: " +libro[prop] );  
}  
  
// Devuelve:  
// La propiedad titulo contiene: Aprendiendo JavaScript  
// La propiedad autor contiene: Carlos Azaustre  
// La propiedad numPaginas contiene: 64  
// La propiedad editorial contiene: carlosazaustre.es  
// La propiedad precio contiene: 2.95
```

JSON

JSON es el acrónimo de *JavaScript Object Notation*. Notación de objeto JavaScript. Es un objeto JavaScript pero con algunos detalles de implementación que nos permitirán serializarlo para poder utilizarlo como intercambio de datos entre servicios. Antes de popularizarse este formato, el más común era XML (*eXtended Marked language*) pero insertaba demasiadas etiquetas HTML (o XML) lo que lo hacía menos legible y más complicado de decodificar. JSON por su parte al ser en sí un objeto JavaScript es ideal para aplicaciones que manejen este lenguaje.

Los detalles de implementación son que las propiedades del objeto deben ser *Strings* para que no haya problemas al codificarlo y decodificarlo.

Debemos tener en cuenta que algunos tipos de datos no se van a serializar igual. Por ejemplo los tipos `NaN` e `Infinity` se codifican como `null`. Y los objetos de tipo `Date` muestran la fecha en formato ISO y al reconstruirlo será un `string` sin poder acceder a los métodos que hereda de la clase `Date`. Tampoco se pueden serializar funciones, expresiones regulares, errores y valores `undefined`. El resto de primitivas y clases como `objects`, `arrays`, `strings`, números, `true`, `false` y `null`.

Veamos un ejemplo de un objeto JSON y como podemos serializarlo y reconstruirlo:

```
var usuario = {
  "id": "012345678",
  "username": "carlosazaustre",
  "password": "fkldfn4r09330adafnanf9843fbcdkjdkks",
  "data": {
    "name": "Carlos Azaustre",
    "email": "carlosazaustre@gmail.com",
    "city": "Madrid",
    "country": "ES"
  },
  "preferences": {
    "contact": {
      "email": true,
      "notify": true
    },
    "interests": [
      "javascript",
      "nodejs",
      "angularjs"
    ]
  }
};
```

Si queremos acceder a una propiedad determinada podemos hacerlo como cualquier objeto:

```
usuario.data.city; // "Madrid"
```

Si queremos serializarlo para poder realizar un intercambio de datos, debemos usar la función `JSON.stringify` que devuelve en un *String* la información del objeto que se le pasa por parámetro.

```
var jsonSerializado = JSON.stringify(usuario);
/* Devuelve:

{"id":"012345678","username":"carlosazaustre","password":"fkldfn4r09330adafnanf9843f
*/
```

Si ahora queremos acceder a las propiedades no podemos, porque se trata de un string.

```
jsonSerializado.data.city;  
/*  
Uncaught TypeError: Cannot read property 'city' of undefined  
    at <anonymous>:2:21  
    at Object.InjectedScript._evaluateOn (<anonymous>:895:140)  
    at Object.InjectedScript._evaluateAndWrap (<anonymous>:828:34)  
    at Object.InjectedScript.evaluate (<anonymous>:694:21)  
*/
```

Para poder reconstruirlo a partir del string, tenemos la función `JSON.parse` que devuelve un objeto a partir del string que se pasa como parámetro. Tiene que estar correctamente formado, si no el método `parse` nos devolverá error.

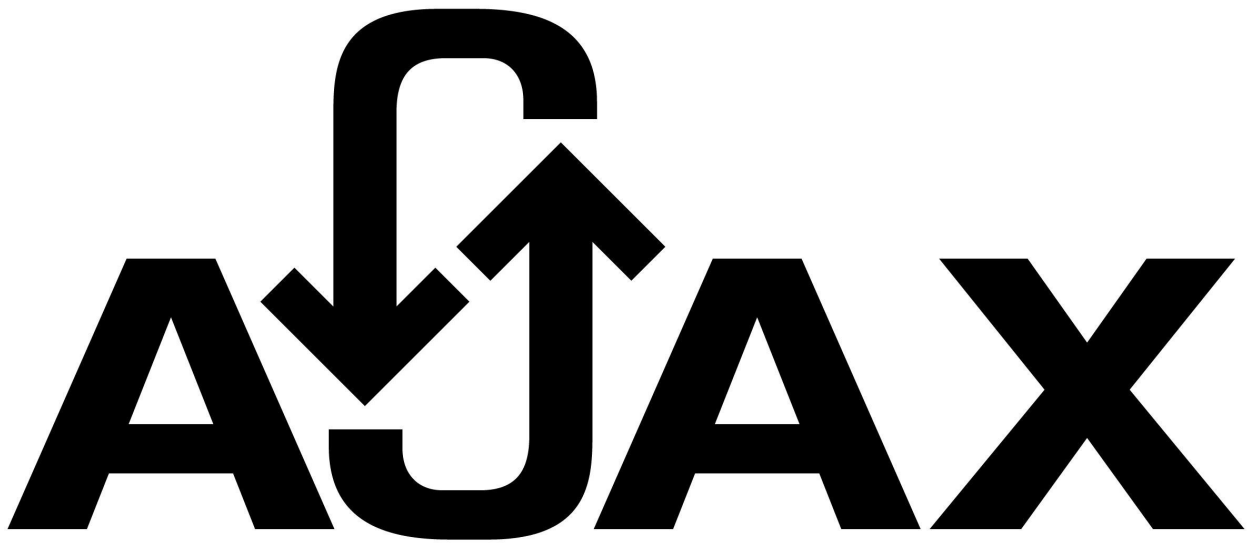
```
var jsonReconstruido = JSON.parse(jsonSerializado);  
/*  
Object {id: "012345678", username: "carlosazaustre", password: "fkldfn4r09330adafnanf"  
*/
```

Ahora podemos acceder a sus propiedades como antes.

```
jsonReconstruido.data.city; // "Madrid"
```

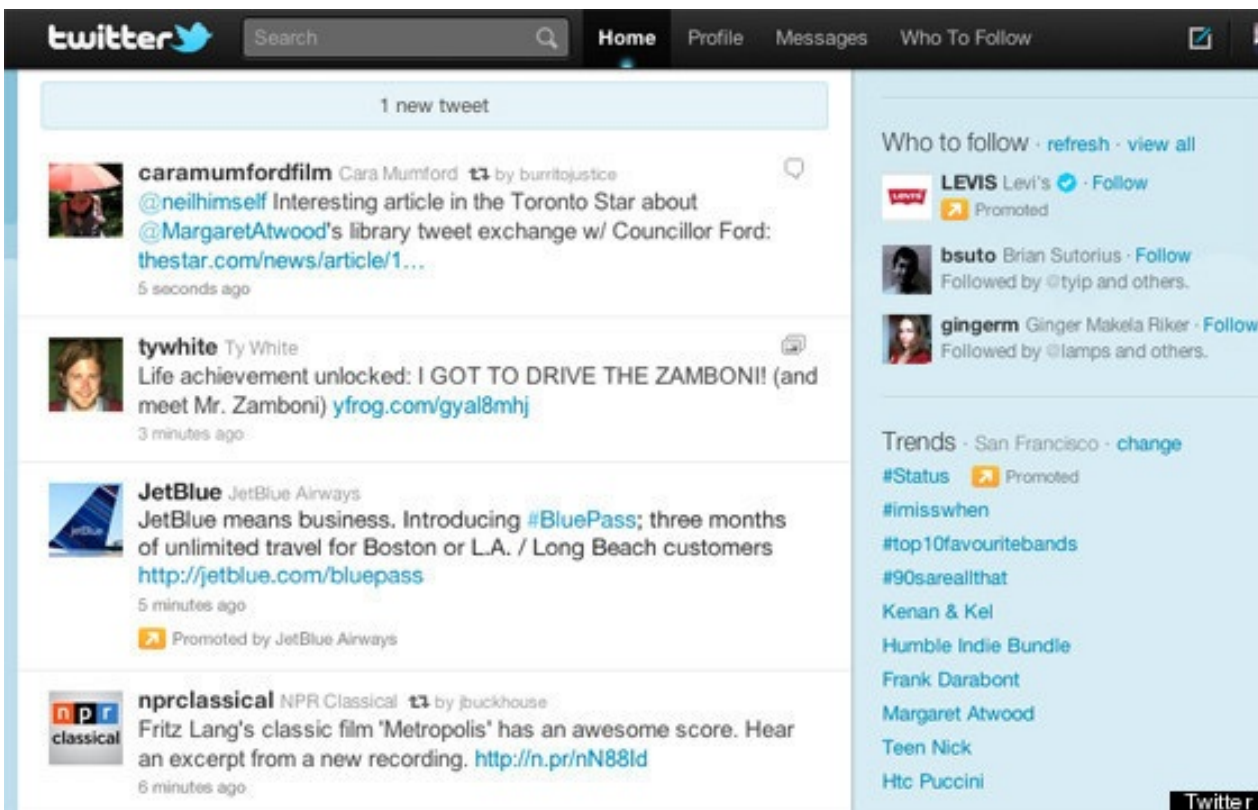
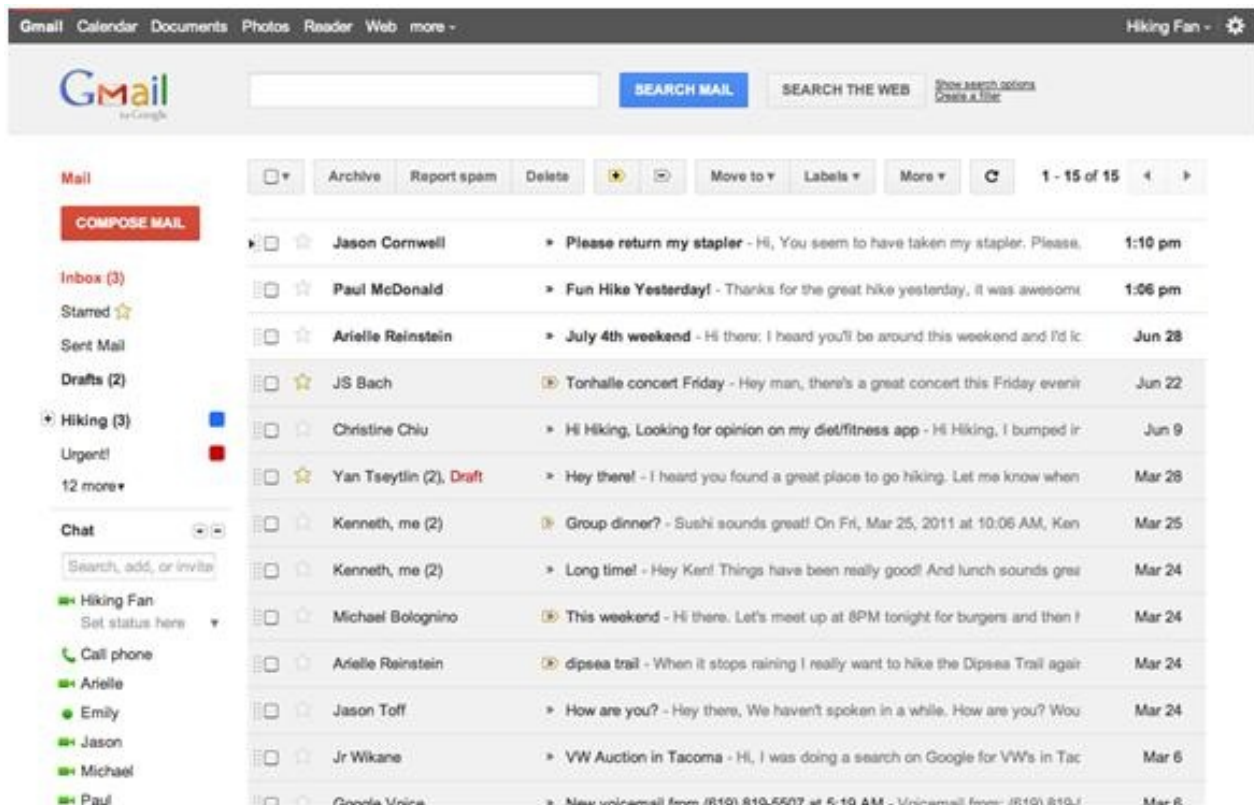
Más adelante veremos dónde se utilizan este tipo de datos, normalmente en intercambios de datos desde un servidor con llamadas HTTP, AJAX, o entre funciones dentro de un mismo programa.

AJAX



AJAX es el acrónimo de *Asynchronous JavaScript And XML*. Es el uso de JavaScript para realizar llamadas o peticiones asíncronas al servidor utilizando XML. La parte de XML ha sido sustituida hoy en día por JSON, que es un formato más amigable para el intercambio de datos, aunque se sigue utilizando la X en el acrónimo AJAX para expresar esta tecnología.

Como dije en la introducción de este libro, JavaScript se hizo muy popular por la llegada de esta tecnología y prueba de ello son las aplicaciones que surgieron y siguen utilizándose hoy en día como Gmail, Twitter, etc...



XMLHttpRequest

Para poder utilizar ésta tecnología hay que hacer uso del objeto `XMLHttpRequest` de JavaScript y tener un servidor que nos devuelva contenido, preferiblemente en formato JSON para poder tratarlo.

Veamos un ejemplo de como utilizar llamadas asíncronas en una página web, imaginemos que tenemos un servidor o nos proporcionan un servicio que a través de una URL nos devuelve una cantidad de datos en formato JSON. por ejemplo la siguiente dirección: `http://jsonplaceholder.typicode.com/photos` , que devuelve lo siguiente:

```
[
  {
    "albumId": 1,
    "id": 1,
    "title": "accusamus beatae ad facilis cum similique qui sunt",
    "url": "http://placeholder.it/600/92c952",
    "thumbnailUrl": "http://placeholder.it/150/30ac17"
  },
  {
    "albumId": 1,
    "id": 2,
    "title": "reprehenderit est deserunt velit ipsam",
    "url": "http://placeholder.it/600/771796",
    "thumbnailUrl": "http://placeholder.it/150/dff9f6"
  },
  {
    "albumId": 1,
    "id": 3,
    "title": "officia porro iure quia iusto qui ipsa ut modi",
    "url": "http://placeholder.it/600/24f355",
    "thumbnailUrl": "http://placeholder.it/150/1941e9"
  },
  ...
]
```

¿Cómo hacemos para llamar a esa URL dentro de nuestra página web de manera asíncrona (sin recargar la página)?


```
// Creamos el objeto XMLHttpRequest
var xhr = new XMLHttpRequest();

// Definimos la función que manejará la respuesta
function reqHandler () {
  if (this.readyState === 4 && this.status === 200) {
    /* Comprobamos que el estado es 4 (operación completada)
     * los estados que podemos comprobar son:
     * 0 = UNSET (No se ha llamado al método open())
     * 1 = OPENED (Se ha llamado al método open())
     * 2 = HEADERS_RECEIVED (Se ha llamado al método send())
     * 3 = LOADING (Se está recibiendo la respuesta)
     * 4 = DONE (se ha completado la operación)
     * y el código 200 es el correspondiente al OK de HTTP de
     * que todo ha salido correcto.
     */
    console.log(this.responseText);
  }
}

// Asociamos la función manejadora
xhr.onload = reqHandler;
// Abrimos la conexión hacia la URL, indicando el método HTTP, en este
// caso será GET
xhr.open('GET', 'http://jsonplaceholder.typicode.com/photos', true);
// Enviamos la petición
xhr.send();

/* Esto es lo que mostrará en la consola:
[
  {
    "albumId": 1,
    "id": 1,
    "title": "accusamus beatae ad facilis cum similique qui sunt",
    "url": "http://placeholder.it/600/92c952",
    "thumbnailUrl": "http://placeholder.it/150/30ac17"
  },
  {
    "albumId": 1,
    "id": 2,
    "title": "reprehenderit est deserunt velit ipsam",
    "url": "http://placeholder.it/600/771796",
    "thumbnailUrl": "http://placeholder.it/150/dff9f6"
  },
  ...
*/
```

Como es un objeto JSON, podemos acceder a sus propiedades, iterarlo, etc.. Entonces imaginemos que el anterior código JavaScript lo tenemos en una página HTML que contiene un `<div id='respuesta'></div>`

```
<!-- Pagina HTML -->
<html>
<body>
  <div id="respuesta"></div>
</body>
</html>
```

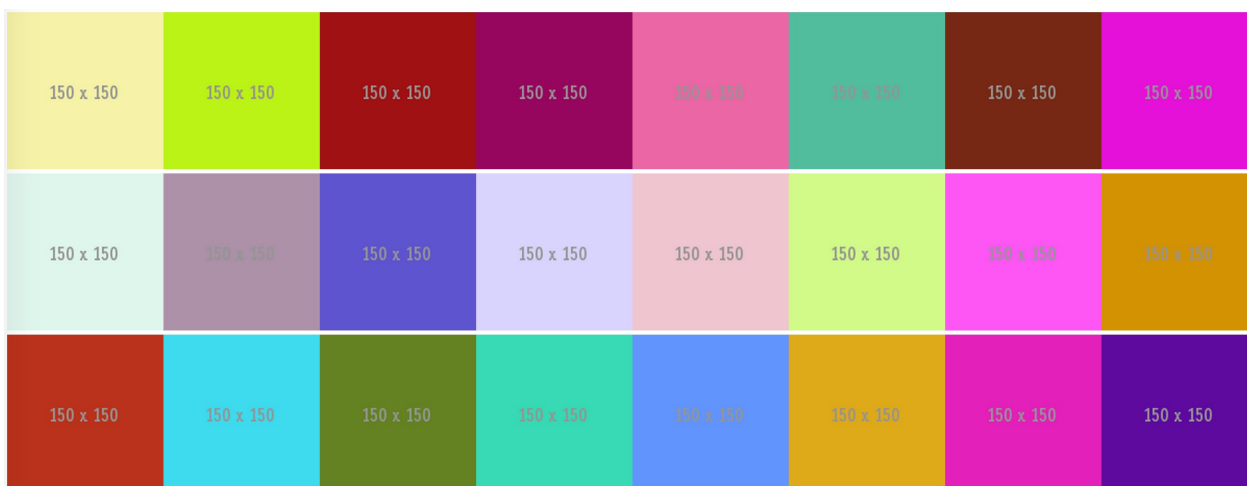
si la función `reqHandler` la modificamos de esta manera:

```
// Código JS
function reqHandler () {
  if (this.readyState === 4 && this.status === 200) {
    var respuesta = JSON.parse(this.responseText);
    var respuestaHTML = document.querySelector('#respuesta');
    var tpl = '';

    respuesta.forEach(function (elem) {
      tpl += '<a href="' + elem.url + '">'
        + '',
        + '</a>',
        + '<br/>',
        + '<span>' + elem.title + '</span>';
    });

    respuestaHTML.innerHTML = tpl;
  }
}
```

Tendríamos este resultado:



Fetch en ECMAScript6

En el nuevo estándar de JavaScript se introduce el objeto `fetch` que funciona de manera similar al objeto `XMLHttpRequest` pero de una forma que resulta más legible y utilizando promesas, recogiendo las funciones con `then`. Todo el tema de promesas lo explicaremos más adelante.

```
var respuestaHTML = document.querySelector('#respuesta');
var tpl = '';

fetch("http://jsonplaceholder.typicode.com/photos" )
  .then((response) => {
    return response.json()
  })
  .then((albums) => {
    albums.forEach(function (elem) {
      tpl += '<a href="' + elem.url + '>'
        + ' {
    return response.json();
  })
  .then((albums) => {
    albums.forEach(function (elem) {
      elemento = document.createTextNode(
        `<a href="${elem.url}">
          
        </a>
        <br/>
        <span>${elem.title}</span>`;
      tpl.appendChild(elemento);
    });
    respuestaHTML.appendChild(tpl);
  });
```

Obtendríamos el mismo resultado que en el anterior ejemplo:



`fetch` por defecto utiliza el método GET de HTTP, si necesitamos una configuración más personalizada, se le puede pasar un objeto con el método, un header y un body para usar con el método POST. También se le puede pasar un modo, unas credenciales de autenticación y parámetros de caché.

```
var myHeaders = new Header();
myHeaders.append('Content-Type', 'application/json');

fetch(URI, {
  method: 'GET',
  headers: myHeaders,
  mode: 'cors',
  cache: 'default'
})
```

El primer `then` devuelve la respuesta, que incluye cabeceras, estado, etc.. Hasta que la respuesta no se haya completado no se puede acceder a los datos. Por eso debemos llamar de nuevo `then` devolviendo la respuesta con `response.json()` entonces es cuando ya podemos manipular el `body` de la respuesta.

Eventos

JavaScript nos permite, por su entorno de programación, una programación orientada a eventos. Podemos detectar eventos que ocurran en el navegador (o en el servidor) y actuar en base a ellos. También podemos crear nuestros propios eventos y suscribirnos, sería lo que se conoce como patrón *PubSub* (Publicador-Suscriptor)

Manejando eventos

Imaginemos que hacemos clic en un elemento HTML de una página web, que no necesariamente sea un enlace, en ese momento se dispara un evento que podemos capturar y realizar la función que estimemos conveniente, una llamada AJAX, un cambio de estilo, etc...

Veamos un ejemplo con código:

```
// Asociamos a un elemento de la web el evento
var target = document.querySelector('#respuesta');
target.addEventListener('click', onClickHandler, false);

// Función que manejará el evento
function onClickHandler(e) {
  e.preventDefault();
  console.log(e);
}
```

la función `e.preventDefault()` evita que se dispare una acción por defecto. Imaginemos que este evento lo estamos realizando sobre un enlace o sobre un botón de un formulario. Gracias a esta función, evitaremos que el enlace nos redirija al hipervínculo o que el botón dispare la acción por defecto del formulario. De esta forma podemos controlar el flujo de la acción en cualquier evento.

En el ejemplo de código anterior, estamos asociando la función `onClickHandler` al evento `click` en un elemento HTML con el *id* `respuesta`. Esto quiere decir que cuando hagamos clic con el ratón en ese elemento HTML, se ejecutará la función, que en el ejemplo hemos puesto mostrará en la consola la información del evento:

```
▼ MouseEvent {} ⓘ  
  altKey: false  
  bubbles: true  
  button: 0  
  buttons: 0  
  cancelBubble: false  
  cancelable: true  
  charCode: 0  
  clientX: 57  
  clientY: 37  
  ctrlKey: false  
  currentTarget: null  
  dataTransfer: null  
  defaultPrevented: false  
  detail: 1  
  eventPhase: 0  
  fromElement: null  
  keyCode: 0  
  layerX: 57  
  layerY: 37  
  metaKey: false  
  movementX: 0  
  movementY: 0  
  offsetX: 49  
  offsetY: 29  
  pageX: 57  
  pageY: 37  
  ▶ path: Array[5]  
    relatedTarget: null  
    returnValue: true  
    screenX: 897  
    screenY: 217  
    shiftKey: false  
  ▶ sourceDevice: InputDevice  
  ▶ srcElement: div#respuesta  
  ▶ target: div#respuesta  
    timeStamp: 1442137686643  
  ▶ toElement: div#respuesta  
    type: "click"  
  ▶ view: Window  
    webkitMovementX: 0  
    webkitMovementY: 0  
    which: 1  
    x: 57  
    y: 37  
  ▶ __proto__: MouseEvent
```

Propagación de eventos

Los eventos pueden propagarse hacia arriba en el documento. En el siguiente ejemplo vamos a escuchar el evento `click` en el elemento `header` que contiene un `h1` y un `h2`

```
<html>
  <body>
    <header>
      <h1>Hola Mundo</h1>
      <h2>SubHeader</h2>
    </header>
  </body>
</html>
```

```
var header = document.querySelector('header');

header.addEventListener('click', function(e) {
  console.log('Has clickado en ' + e.target.nodeName);
});
```

Con el manejador creado, se imprimirá en consola el mensaje `Has clicado en` seguido del nombre del elemento gracias a `e.target.nodeName`. Si clicamos dentro del `h1` nos mostrará `Has clickado en H1` y si lo hacemos en el `h2` mostrará `Has clicado en H2`.

Aunque estemos escuchando el elemento `header`, tenemos acceso a todos los nodos que se encuentren dentro de el.

Ahora imaginemos que también añadimos un escuchador de eventos al documento raíz `document` como el siguiente:

```
document.addEventListener('click', function(e) {
  console.log('Has clickado en el documento');
});
```

Cuando hagamos clic en cualquier parte de la página, nos mostrará el mensaje `Has clicado en el documento`. Pero si clicamos en una parte del `header` tendremos los dos mensajes por consola:


```
Has clicado en el documento
Has clicado en H1
```

Si queremos mantener el escuchador en el `document` pero cuando hagamos clic en `header` no salte el otro evento, podemos hacer uso de `e.stopPropagation()`, para evitar que se propague de abajo a arriba.

```
header.addEventListener('click', function(e) {
  console.log('Has clicado en ' + e.target.nodeName);
  e.stopPropagation();
});
```

De esta forma si clicamos en `h1` o en `h2` obtendremos `Has clicado en HX`, y sin que aparezca el evento asociado a `document`.

...

Tenemos a nuestra disposición numerosos eventos sobre los que podemos actuar. En este [enlace](#) tienes la lista completa. Los más utilizados en una aplicación web serían:

```
- click, dblclick
- change
- drag (dragenter, dragend, dragleave, dragover, dragstart,...)
- focus
- keydown, keyup, keypress,
- mouseenter, mouseleave, mouseover, mouseup,...
- scroll
- submit
...
```

Y a cualquiera de estos eventos, le podemos asociar una función manejadora que realizará lo que programemos.

Patrón PubSub

Además de los eventos que nos proporciona el DOM, podemos crear los nuestros propios. Esto se le conoce como el patrón *PubSub*. Realizamos una acción y publicamos o emitimos un evento. En otra parte de nuestro código escuchamos ese evento y cuando se produzca realizamos otra acción. Veamos un ejemplo con código.

Vamos a crear un *closure* llamado `pubsub` donde tendremos 2 funciones, la función `subscribe` donde escucharemos los eventos, y la función `publish` que los publicará

```
var pubsub = (function() {
  // Este objeto actuará como cola de todos los eventos que se
  // produzcan. Los guardará con el nombre del evento como clave
  // y su valor será un array con todas las funciones callback encoladas.
  var suscriptores = {};

  function subscribe(event, callback) {
    // Si no existe el evento, creamos el objeto y el array de callbacks
    // y lo añadimos
    if (!suscriptores[event]) {
      var suscriptorArray = [callback];
      suscriptores[event] = suscriptorArray;

      // Si existe, añadimos al array de callbacks la función pasada por
      // parámetro
    } else {
      suscriptores[event].push(callback);
    }
  }

  function publish(event) {
    // Si el evento existe, recorremos su array de callbacks y los
    // ejecutamos en orden.
    if (suscriptores[event]) {
      suscriptores[event].forEach(function (callback) {
        callback();
      });
    }
  }

  return {
    // Los métodos públicos que devolvemos serán `pub` y `sub`
    pub: publish,
    sub: subscribe
  };
})();
```

Por tanto, para escuchar un evento y realizar una operación asociada, deberemos llamar a `pubsub.sub`, pasarle como primer parámetro el nombre del evento, en este caso le vamos a llamar `miEvento`, seguido de una función manejadora. En este caso simplemente vamos a imprimir por consola que el evento se ha disparado.

```
pubsub.sub('miEvento', function(e) {  
  console.log('miEvento ha sido lanzado!');  
});
```

Para poder lanzar el evento y que posteriormente sea recogido por la función `pubsub.sub`, lo emitimos con la función `pubsub.pub` pasándole como parámetro el nombre del evento. En este caso `miEvento`:

```
pubsub.pub('miEvento');
```

Esto mostrará en la consola lo siguiente:

```
miEvento ha sido lanzado!
```

Patrón Pub/Sub con Datos.

Además de emitir y escuchar el evento, podemos pasar un objeto con datos en la operación, y así utilizarlo a lo largo de nuestra aplicación.

Por ejemplo, queremos que al emitir un evento, poder pasar un objeto de la siguiente manera:

```
pubsub.pub('MiEvento', {  
  misDatos: 'Estos son mis datos'  
});
```

Y al escucharlo, poder mostrarlo:

```
pubsub.sub('MiEvento', function(e) {  
  console.log('miEvento ha sido lanzado, y contiene: ', e.data.misDatos);  
});
```

Para lograrlo, debemos modificar un poco la función `pubsub` creando un objeto dónde almacenaremos los datos que publiquemos. Nuestro *closure* `pubsub` quedaría así:

```
var pubsub = (function() {  
  var suscriptores = {};  
  
  function EventObject() {};  
  
  function subscribe(event, callback) {  
    if (!suscriptores[event]) {  
      var suscriptorArray = [callback];  
      suscriptores[event] = suscriptorArray;  
    } else {  
      suscriptores[event].push(callback);  
    }  
  }  
  
  function publish(event, data) {  
    var eventObject = new EventObject();  
    eventObject.type = event;  
  
    if (data) {  
      eventObject.data = data;  
    }  
  
    if (suscriptores[event]) {  
      suscriptores[event].forEach(function (callback) {  
        callback(eventObject);  
      });  
    }  
  }  
  
  return {  
    pub: publish,  
    sub: subscribe  
  };  
})();
```

Y con todo ésto, obtendríamos en la consola:

```
miEvento ha sido lanzado y contiene  Estos son mis datos!
```

WebSockets

Los websockets son una tecnología que permite una comunicación bidireccional (en ambos sentidos) entre el cliente y el servidor sobre un único socket TCP. En cierta manera es un buen sustituto de AJAX como tecnología para obtener datos del servidor, ya que no tenemos que pedirlos, el servidor nos los enviará cuando haya nuevos.

Otra de las ventajas que tiene es el tamaño del mensaje. En un mensaje enviado por HTTP (con GET, POST, etc...) el tamaño de la cabecera pesa alrededor de 100 bytes, y si el mensaje es enviado por websockets, su cabecera tiene un tamaño de 2 bytes, lo que lo hace más ligero y por tanto más rápido.

Uno de los ejemplos más comunes para aprender a utilizar websockets, es desarrollando una aplicación chat. Lo único que necesitaremos para que funcione, es un servidor de websockets, que construiremos en Node.js con la librería [Socket.io](#) que nos facilita el desarrollo de aplicaciones utilizando Websockets en el cliente y en el servidor.

Con tres ficheros de pocas líneas tendremos la funcionalidad básica de un chat implementada. Serían por la parte servidor un fichero que llamaremos `server/main.js` y por la parte cliente, `public/main.js` y el HTML en `public/index.html`.

El `index.html` es muy sencillo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>My Aplicación con Sockets</title>
  <script src="/socket.io/socket.io.js"></script>
  <script src="main.js"></script>
</head>
<body>
  <h1>My App</h1>
  <div id="messages"></div>
  <br/>
  <form onsubmit="return addMessage(this)">
    <input type="text" id="username" placeholder="Tu Nombre" >
    <input type="text" id="texto" placeholder="Cuéntanos algo...">
    <input type="submit" value="Enviar!">
  </form>
</body>
</html>
```

Simplemente tiene un `div` con el id `messages` que es dónde se pintarán los mensajes de chat a través de JavaScript. También tiene un formulario para enviar nuevos mensajes al servidor y al resto de clientes conectados. Como puntualización, fijarse que tenemos el `script` de Socket.io referenciado, al igual que el de nuestro fichero `public/main.js`.

Veamos la funcionalidad del lado cliente:

```
var socket = io.connect('http://localhost:8080', { 'forceNew': true });

socket.on('messages', function(data) {
  console.log(data);
  render(data);
})

function render (data) {
  var html = data.map(function(elem, index) {
    return(`<div>
      <strong>${elem.author}</strong>:
      <em>${elem.text}</em>
    </div>`);
  }).join(" ");

  document.getElementById('messages').innerHTML = html;
}

function addMessage(e) {
  var message = {
    author: document.getElementById('username').value,
    text: document.getElementById('texto').value
  };

  socket.emit('new-message', message);
  return false;
}
```

En este archivo, primero de todo, creamos un socket con socket.io conectándolo a nuestro servidor de websockets que tendremos corriendo en `http://localhost:8080` y que veremos en unos momentos.

Dejaremos escuchando a ese socket el evento o mensaje `messages` con `socket.on('messages')` y a continuación en su función de callback lo renderizamos en el HTML.

Para renderizarlo en el HTML hemos creado una función `render(data)`. Esta función simplemente toma los datos que le llegan a través del socket (Que será un array de mensajes) y con la función `map`, los itera y pinta una plantilla HTML con el nombre del autor del mensaje y el texto del mismo.

Como particularidad, para el String de salida hemos utilizado una nueva funcionalidad de ECMAScript6 que se llama *Template String* que permite interpolar variables utilizando los operadores `${nombre_variable}` sin necesidad de tener que concatenar

Strings con el operador más `+`. También nos permite escribir en varias líneas también sin concatenar. Por último, en esta función empleamos el método `join` para unir los elementos del array con espacios, en lugar de la coma que trae por defecto.

También en este archivo tenemos implementada la función `addMessage` que se dispara cuando pulsamos el botón de submit del formulario de envío de mensajes. Lo que realiza esta función es recoger el valor de los input del formulario, el que tiene el id `author` y el del id `text`, para por último enviar por sockets con el mensaje `new-message` esos datos.

En resumen, nuestra aplicación cliente escucha el evento o mensaje `messages` para recibir datos y tratarlos, y emite el evento o mensaje `new-message` para enviar los nuevos. Estos eventos deben estar también implementados en el servidor, y eso es lo que veremos ahora en el fichero `server/main.js`:

```
var express = require('express');
var app = express();
var server = require('http').Server(app);
var io = require('socket.io')(server);

var messages = [{
  text: "Hola soy un mensaje",
  author: "Carlos Azaustre"
}];

app.use(express.static('public'));

io.on('connection', function(socket) {
  console.log('Alguien se ha conectado con Sockets');
  socket.emit('messages', messages);

  socket.on('new-message', function(data) {
    messages.push(data);

    io.sockets.emit('messages', messages);
  });
});

server.listen(8080, function() {
  console.log("Servidor corriendo en http://localhost:8080");
});
```

Este es un servidor escrito en Node.js, y es muy sencillo. En las primeras líneas lo que tenemos son las librerías que estamos importando como son Express que es un framework para facilitarnos el crear aplicaciones web en Node.js, y Socket.io para el

tratamiento de los websockets.

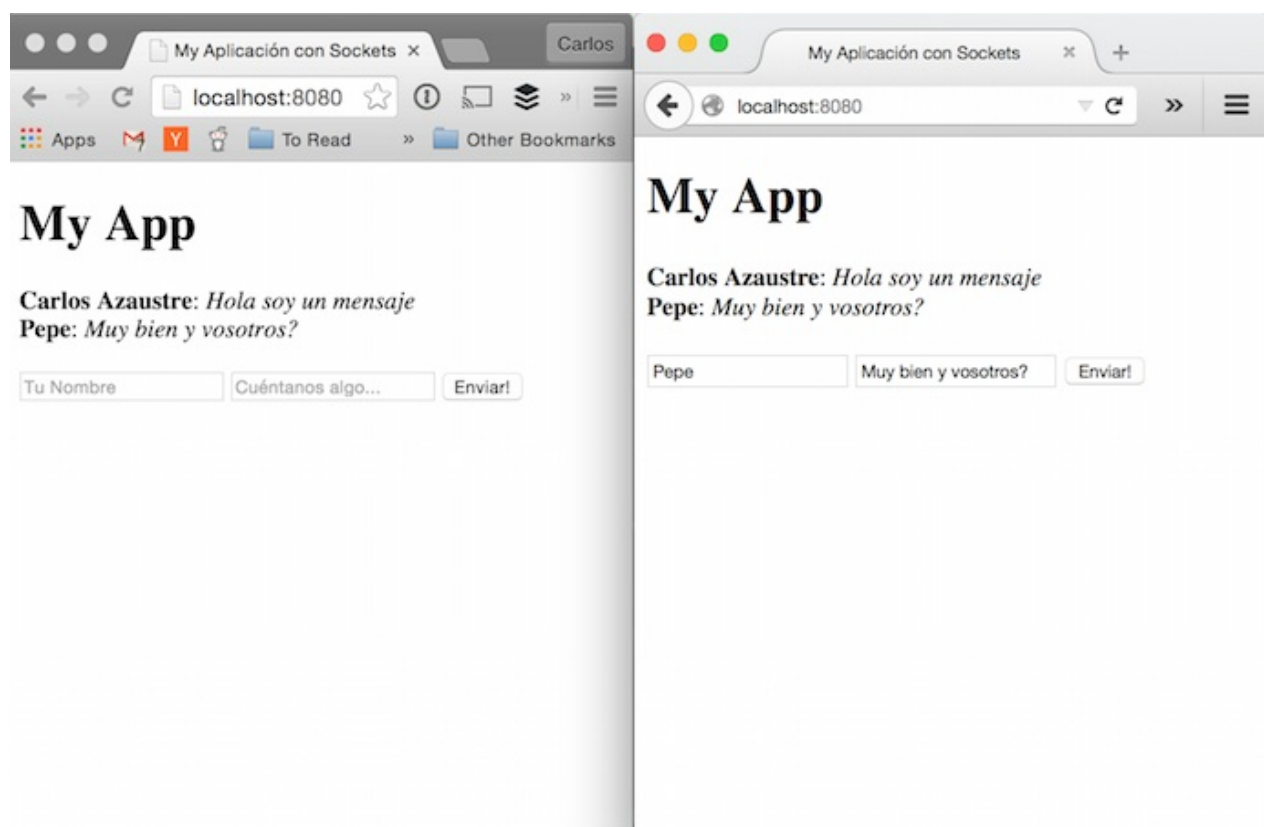
Creamos una aplicación `Express` y un servidor `http` al que le pasamos la aplicación, seguidamente importamos `Socket.io` en la variable `io` y le pasamos el servidor creado. Éste será nuestro servidor de websockets.

la variable `messages` es un array en el que tenemos por ahora un sólo elemento. Aquí almacenaremos los mensajes que se produzcan en el chat. Esto solo es por motivos didácticos. Lo normal sería que los mensajes se almacenaran en una base de datos, pero ahora mismo sería complicar mucho el ejemplo.

Usamos el middleware de `Express` `express.static` para indicar a nuestro servidor `Node.js` cuál es la ruta para los ficheros estáticos o la parte cliente de nuestra aplicación, la cuál tenemos en la carpeta `public`.

Ahora ponemos a escuchar a nuestro servidor de websockets con `io.on()` y le pasamos el mensaje `connection`. Este mensaje le llegará cuando la aplicación cliente se lance. Dentro de esta función podemos hacer uso de los websockets. En concreto usamos uno para emitir el evento `messages` y en el cuál pasamos el array de mensajes que tenemos almacenado. Esto llegará al cliente que está escuchando este evento.

Después escuchamos el evento que lanza el cliente, el evento `new-message` con `socket.on('new-message')` y aquí lo que hacemos es obtener los datos desde el cliente (el objeto mensaje) y añadirlos al array `messages` con el método `push`. Seguidamente, emitimos desde un socket a todos los clientes conectados con el método `io.sockets.emit` un evento `messages` al que pasamos el array de mensajes. De esta manera todos los sockets o clientes conectados a la aplicación recibirán en tiempo real los datos (es decir, los mensajes del chat) sin necesidad de recargar la página. Si lo hicieramos mediante AJAX tradicional, para ver si hay mensajes nuevos deberíamos recargar la página o disparar un evento ya sea pulsando un botón en el HTML o estableciendo un `timeout`.



Para este tipo de aplicaciones en tiempo real, como pueden ser un chat, un videojuego, etc... lo conveniente es utilizar websockets.

Promesas

Introducción

Una Promesa en JavaScript, es un objeto que sirve para reservar el resultado de una operación futura.

Este resultado llega a través de una operación asíncrona como puede ser una petición HTTP o una lectura de ficheros, que son operaciones no instantáneas, que requieren un tiempo, aunque sea pequeño, para ejecutarse y finalizar.

Para entender mejor el concepto de Promesas, veremos primero como funciona la *asincronía* en JavaScript con la típica función de callback

```
function loadCSS(url, callback) {  
  var elem = window.document.createElement('link');  
  elem.rel = "stylesheet";  
  elem.href = url;  
  window.document.head.appendChild(elem);  
  callback();  
}  
  
loadCSS('styles.css', function() {  
  console.log("Estilos cargados");  
});
```

En este ejemplo tenemos una función llamada `loadCSS` a la que pasamos una `url`, presumiblemente que apunte a un fichero `.css` y una función de callback como parámetros, la función básicamente crea un elemento `link` y lo añade al final de la etiqueta `<head>`

Cuando ejecutamos esta función, le pasamos la url de `styles.css` y una función anónima que será el callback. Lo que hará será imprimir por la consola `Estilos cargados` cuando finalice la carga.

Este es un ejemplo básico de una función asíncrona con callbacks. Si queremos reproducir este mismo comportamiento usando Promesas, sería de la siguiente manera:

```
// Asumamos que loadCSS devuelve una promesa
var promise = loadCSS('styles.css');

promise.then(function() {
  console.log("Estilos cargados");
});

promise.catch(function(err) {
  console.log("Ocurrió un error: " + err);
});
```

Si `loadCSS` fuese una función asíncrona que devuelve una promesa, la resolveríamos utilizando la función `then`. Esta función se ejecuta cuando la promesa ha sido resuelta. Si hubiese ocurrido algún error, se ejecutaría la función `catch`, donde recogemos el error y lo tratamos.

La función la guardamos en la variable `promise`. Como es posible aplicar "chaining", es decir, anidamiento de funciones, podemos escribir la ejecución de la siguiente manera, que es más legible y elegante:

```
loadCSS('styles.css')
  .then(function() {
    console.log("Estilos cargados");
  })
  .catch(function(err) {
    console.log("Ocurrió un error: " + err);
  });
```

Pero para que se puedan usar las funciones `then` y `catch`, hay que implementar la función `loadCSS` como una promesa.

Librerías de terceros para utilizar Promesas

En ECMAScript5 tenemos varias librerías que nos permiten hacer eso de una forma cómoda, una de las más utilizadas es Q. Así sería con esta librería y utilizando ES5:

```
var Q = require('q');

function loadCSS(url) {
  var deferred = Q.defer();

  var elem = window.document.createElement('link');
  elem.rel = "stylesheet";
  elem.href = url;
  window.document.head.appendChild(elem);
  deferred.resolve();

  return deferred.promise;
}
```

El procedimiento consiste en crear un objeto `deferred` con la librería `q`. De seguido realizamos las operaciones de nuestra función, en este caso la creación del elemento `link` y su anidamiento al `<head>`. Cuando finalicemos, ejecutamos la función `resolve()` del objeto `deferred` que le indica a la promesa que devolvemos que todo ha ido bien.

Si por algún motivo tenemos que comprobar si el resultado que vamos a devolver es correcto o no, tenemos la función `deferred.reject()` que lanzará la función `catch` de la promesa.

Por ejemplo, imagina una función división entre dos valores

```
var Q = require('q');

function division (num1, num2) {
  var deferred = Q.defer();

  if (num2 == 0) {
    deferred.reject(new Error("Dividir entre cero es Infinito"));
  } else {
    deferred.resolve(num1 / num2);
  }

  return deferred.promise;
}

division(2,1)
  .then(function(result) {
    console.log("El resultado es: " + result);
  })
  .catch(function(err) {
    console.log("Ha ocurrido un error: " + err);
  })
```

Si dividimos 2 entre 1, el resultado será correcto y nos devolverá el resultado `resolve` de la promesa en la función `then`. Si hubiésemos pasado un 0 como segundo parámetro, el resultado de la promesa hubiera sido el error que lanza `reject` y lo capturaría la función `catch`.

Promesas nativas en ES6

Con el nuevo estándar ECMAScript6, no necesitamos importar una librería de terceros. ES6 trae de forma nativa un nuevo API para Promesas.

Los ejemplos anteriores, implementados con ECMAScript6 serían así:

```
// Para la función loadCSS
function loadCSS (url) {
  var promise = new Promise(
    function resolve(resolve, reject) {
      var elem = window.document.createElement('link');
      elem.rel = "stylesheet";
      elem.href = url;
      window.document.head.appendChild(elem);

      resolve();
    }
  );

  return promise;
}

// Para la función division
function division (num1, num2) {
  var promise = new Promise(
    function resolver(resolve, reject) {
      if (num2 == 0) {
        reject(new Error("Dividir entre cero es Infinito"));
      } else {
        resolve(num1 / num2);
      }
    }
  );

  return promise;
}
```

No te preocupes si este concepto te parece difícil, porque lo es. Es costoso de comprender, pero a la larga te acabas acostumbrando y lo encuentras cómodo, sobre todo en ES6 donde ya viene por defecto de forma nativa.

DOM - Document Object model

El DOM es un conjunto de utilidades específicamente diseñadas para manipular documentos XML, y por tanto documentos HTML.

El DOM transforma el archivo HTML en un árbol de nodos jerárquico, fácilmente manipulable.

Los nodos más importantes son:

- **Document:** Representa el nodo raíz. Todo el documento HTML.
- **Element:** Representa el contenido definido por un par de etiquetas de apertura y cierre, lo que se conoce como un tag HTML. Puede tener como hijos más nodos de tipo `Element` y también atributos.
- **Attr:** Representa el atributo de un elemento.
- **Text:** Almacena el contenido del texto que se encuentra entre una etiqueta HTML de apertura y cierre.

Recorriendo el DOM

Para poder recorrer el DOM, lo podemos hacer a través de un API JavaScript con métodos para acceder y manipular los nodos. Algunas de estas funciones son:

- `getElementById(id)` : Devuelve el elemento con un `id` específico.
- `getElementsByName(name)` : Devuelve los elementos que un `name` (nombre) específico.
- `getElementsByTagName(tagname)` : Devuelve los elementos con un nombre de `tag` específico.
- `getElementsByClassName(classname)` : Devuelve los elementos con un nombre de clase específico.
- `getAttribute(attributeName)` : Devuelve el valor del atributo con nombre `attributeName`
- `querySelector(selector)` : Devuelve un único elemento que corresponda con el `selector` , ya sea por tag, id, o clase.
- `querySelectorAll(selector)` : Devuelve un array con los elementos que correspondan con la query introducida en `selector`

Manipulando el DOM

De igual manera podemos manipular el DOM con las siguientes funciones

- `createElement(name)` : Crea un elemento HTML con el nombre que le pasemos en el parámetro `name` .
- `createTextNode(text)` : Crea un nodo de texto que puede ser añadido a un elemento HTML.
- `createTextAttribute(attribute)` : Crea un atributo que puede ser añadido posteriormente a un elemento HTML.
- `appendChild(node)` : Nos permite *hacer hijo* un elemento a otro.
- `insertBefore(new, target)` : Permite insertar un elemento o nodo `new` antes del indicado en `target` .
- `removeAttribute(attribute)` : Elimina el atributo de nombre `attribute` del nodo desde el que se le llama.
- `removeChild(child)` : Elimina el nodo hijo que se indica con `child`
- `replaceChild(new, old)` : Reemplaza el nodo `old` por el que se indica en el parámetro `new` .

Propiedades de los nodos del DOM

Todos los nodos tienen algunas propiedades que pueden ser muy útiles para las necesidades de nuestros desarrollos:

- `attributes` : Nos devuelve un objeto con todos los atributos que posee un nodo.
- `className` : Permite setear o devolver el nombre de la clase (para CSS) que tenga el nodo si la tiene.
- `id` : Igual que `className` pero para el atributo `id`
- `innerHTML` : Devuelve o permite insertar código HTML (incluyendo tags y texto) dentro de un nodo.
- `nodeName` : Devuelve o nombre del nodo, si es un `<div>` devolverá `DIV` .
- `nodeValue` : Devuelve el valor del nodo. Si es de tipo `element` devolverá `null` . Pero por ejemplo si es un nodo de tipo texto, devolverá ese valor.
- `style` : Permite insertar código CSS para editar el estilo.
- `tagName` : Devuelve el nombre de la etiqueta HTML correspondiente al nodo. Similar a `nodeName`, pero solo en nodos de tipo tag HTML.
- `title` : Devuelve o permite modificar el valor del atributo `title` de un nodo.
- `childNodes` : Devuelve un array con los nodos hijos del nodo desde el que se llama.

- `firstChild` : Devuelve el primer hijo.
- `lastChild` : Devuelve el último hijo.
- `previousSibling` : Devuelve el anterior "hermano" o nodo al mismo nivel.
- `nextSibling` : Devuelve el siguiente "hermano" o nodo al mismo nivel.
- `ownerDocument` : Devuelve el nodo raíz donde se encuentra el nodo desde el que se llama.
- `parentNode` : Devuelve el nodo padre del nodo que se llama.

Vemos a continuación un ejemplo de código que utilice varias cosas de las que hemos visto en teoría:

```
var newElem = document.createElement('div')
newElem.id = 'nuevoElemento'
newElem.className = 'bloque'
newElem.style = 'background:red; width:200px; height:200px'
var body = document.querySelector('body')
body.appendChild(newElem)
```

```
<!-- Resultado: -->
<body>
  <div id="nuevoElemento" class="bloque" style="background:red; width:200px; height:200px">
</body>
```

Este ejemplo lo que hace es crea un elemento `div` con un `id` de nombre `nuevoElemento`, una clase `bloque` y un estilo CSS que define un color de fondo `red` (rojo) y un ancho y alto de 200px.

Todo el API del DOM nos permite cualquier modificación y edición de elementos HTML, de forma que dinámicamente, por ejemplo por medio de eventos, podemos modificar el aspecto y funcionalidad del documento HTML que estamos visualizando.

¿Qué trae nuevo ECMAScript 6?

ECMAScript v6 es la nueva versión del estándar que rige el lenguaje JavaScript. Como te hablé en el primer artículo sobre la historia de JavaScript, la versión que hemos estado empleando hasta ahora en los navegadores era la de ECMAScript 5.1.

La nueva versión también es llamada ES6 para simplificar o ECMAScript 2015 (o ES2015) para dejar claro el año de su aprobación.

Esta nueva versión trae varias novedades, muchas de ellas hacen que el lenguaje se parezca más a otros tipo Python o Java para así atraer a programadores de diferentes stacks. En este capítulo te muestro las más importantes:

Función Arrow

Este tipo de funciones nos permite simplificar sobretodo las típicas funciones *callback* y/o anónimas. Por ejemplo, imagina el siguiente código bastante habitual en JavaScript:

```
var datos = [4, 8, 15, 16, 23, 42];
datos.forEach(function (num) {
  console.log(num)
})
```

Con ES6 y las funciones arrow `=>` podemos ahorrarnos escribir la palabra `function` y también tendremos *bindeado* o enlazado el objeto `this` que anteriormente teníamos que hacerlo de forma manual.

En ES6 esta sería la traducción:

```
var datos = [4, 8, 15, 16, 23, 42];
datos.forEach((num) => {
  console.log(num)
})
```

De hecho, si la función tiene un sólo parámetro, podemos ahorrarnos el paréntesis, quedando así:

```
var datos = [4, 8, 15, 16, 23, 42];
datos.forEach(num => {
  console.log(num)
})
```

He incluso se puede aún simplificar más, ya que la función de callback solamente tiene una línea podemos prescindir de las llaves y dejarlo todo en una única línea:

```
var datos = [4, 8, 15, 16, 23, 42];
datos.forEach(num => console.log(num));
```

Por supuesto, podemos utilizar las *Arrow Functions* al definir funciones, no únicamente en los callbacks. Por ejemplo:

```
// ES5
var saludar = function () {
  console.log('Hola Mundo!')
}
// ES6
var saludar = () => console.log('Hola Mundo!')
```

Objeto `this`

Como comentaba anteriormente, con las funciones `arrow` es más fácil y comprensible enlazar el objeto `this`. ¿Cómo funciona el objeto `this`? Veámoslo con un ejemplo, cuando estamos en una función, tenemos un contexto. Si dentro de ésta función, tenemos otra función, el contexto será diferente. `this` hace referencia al contexto en el que nos encontramos, por tanto el `this` de la función primera, no será el mismo `this` que el de la función de dentro.

En el siguiente código, el objeto `obj` tiene un contexto, que es `this`. En el podemos acceder a la función `foo()` y `bar()`. Dentro de `bar()` si queremos acceder al objeto `this` anterior, no podemos simplemente llamar a `this.foo()` porque no lo reconocerá, ya que en esa función, `this` tiene una referencia diferente. Y más aún cuando realizamos la llamada a `document.addEventListener` y dentro de la función de callback queremos llamar a `this.foo`.

Para conseguir eso, hay varias estrategias dependiendo de la versión de JavaScript que estemos utilizando. En el primer código empleamos la versión 3 de ECMAScript. Entonces la única alternativa que tenemos es *cachear* el objeto `this` en otra variable

(Por ejemplo `that`) y utilizarlo cuando queramos

```
// ES3
var obj = {
  foo : function() {...},
  bar : function() {
    var that = this;
    document.addEventListener("click", function(e) {
      that.foo();
    });
  }
}
```

En la versión 5.1 de ECMAScript, tenemos a nuestra disposición la función `bind` que permite *enlazar* el objeto que le pasemos, en este caso `this` . De esta manera tenemos el mismo comportamiento que en el código anterior pero de una forma más elegante sin tener que *cachear* variables. Solo hay que emplear la función `bind` en la función en la que queramos usar el objeto.

```
// ES5
var obj = {
  foo : function() {...},
  bar : function() {
    document.addEventListener("click", function(e) {
      this.foo();
    }).bind(this));
  }
}
```

Y por último, en ECMAScript 6, haciendo uso de la funciones *Arrow* ya no es necesario hacer nada más. El enlace al objeto `this` va implícito.

```
// ES6
var obj = {
  foo : function() {...},
  bar : function() {
    document.addEventListener("click", (e) => this.foo());
  }
}
```

Template Strings

Una nueva forma de imprimir *Strings* interpolando datos variables sin necesidad de tener que concatenar varios textos. Cuando antes hacíamos algo como esto:

```
// ES5
var dato1 = 1
var dato2 = 2
console.log('La suma de ' + dato1 + ' y de ' + dato2 + ' es ' + dato1 + dato2)
```

Ahora lo podemos hacer más fácil utilizando el acento ``` y el operador `${}` para encapsular las variables:

```
// ES6
var dato1 = 1
var dato2 = 2
console.log(`La suma de ${dato1} y de ${dato2} es ${dato1 + dato2}`)
```

También nos permite imprimir strings multilínea sin necesidad de concatenar. Lo que antes era así:

```
// ES5
var template = '<div>' +
  '<ul>' +
    '<li>Item 1</li>' +
    '<li>Item 2</li>' +
    '<li>Item 3</li>' +
  '</ul>' +
  '</div>'
```

Con ECMAScript 6 lo podemos escribir de una manera más rápida utilizando el acento ``` al inicio del String y al final, insertando todos los saltos de línea deseados, en lugar de las comillas dobles o simples para escribirlo:

```
// ES6
var template = `<div>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</div>`
```

let y const

Ahora podemos declarar variables con la palabra reservada `let` en lugar de `var` para definirla únicamente dentro de un bloque. Ya que con `var`, si la variable no estaba definida dentro de un *Closure*, quedaba definida de manera global:

```
//ES5
(function() {
  console.log(x); // x no está definida aún.
  if (true) {
    var x = "hola mundo";
  }
  console.log(x);
  // Imprime "hola mundo", porque "var" hace que sea global
  // a la función;
})();

//ES6
(function() {
  if (true) {
    let x = "hola mundo";
  }
  console.log(x);
  //Da error, porque "x" ha sido definida dentro del bloque "if"
})();
```

Módulos

Desde hace mucho se pedía *a gritos* un sistema de módulos que hiciera de JavaScript un lenguaje más potente y manejable. Con Node.js podemos importar módulos con la función `require('nombre_del_modulo')` y gracias a *Browserify* podemos usar esa misma técnica con la parte cliente.

También teníamos alternativas como *RequireJS* e incluso el framework *AngularJS* trae su propio sistema modular.

Sin embargo no existía ninguna forma nativa de utilizar módulos. Ahora con ECMAScript6 por fin es posible.

Exportando e importando módulos

Para exportar un módulo y que éste sea visible por el resto de la aplicación, lo hacemos con la expresión `export nombre_del_modulo`. Veamos un ejemplo:

```
class Coche {
  constructor (nombre, tipo) {
    this.nombre = nombre
    this.tipo = tipo
  }

  arrancar () {
    console.log(`Arrancando el ${this.nombre}`)
  }
}

export default Coche
```

Cómo únicamente tenemos una clase o función en este fichero, lo podemos exportar con `default`. De esta manera si en otra parte de la aplicación queremos importarlo lo podemos hacer como el siguiente ejemplo:

```
import Coche from './coche'
let ford = new Coche('Ford', '5 puertas')
ford.arrancar() // Arrancando el Ford
```

En cambio, si en un mismo fichero tenemos varias funciones o clases que exportar, no podemos utilizar 'default' si no que habría que exportarlo como en el siguiente ejemplo:

```
function caminar () { ... }
function correr () { ... }

export caminar
export correr
```

Y para importarlos desde otro fichero, el `import` cambia un poco:

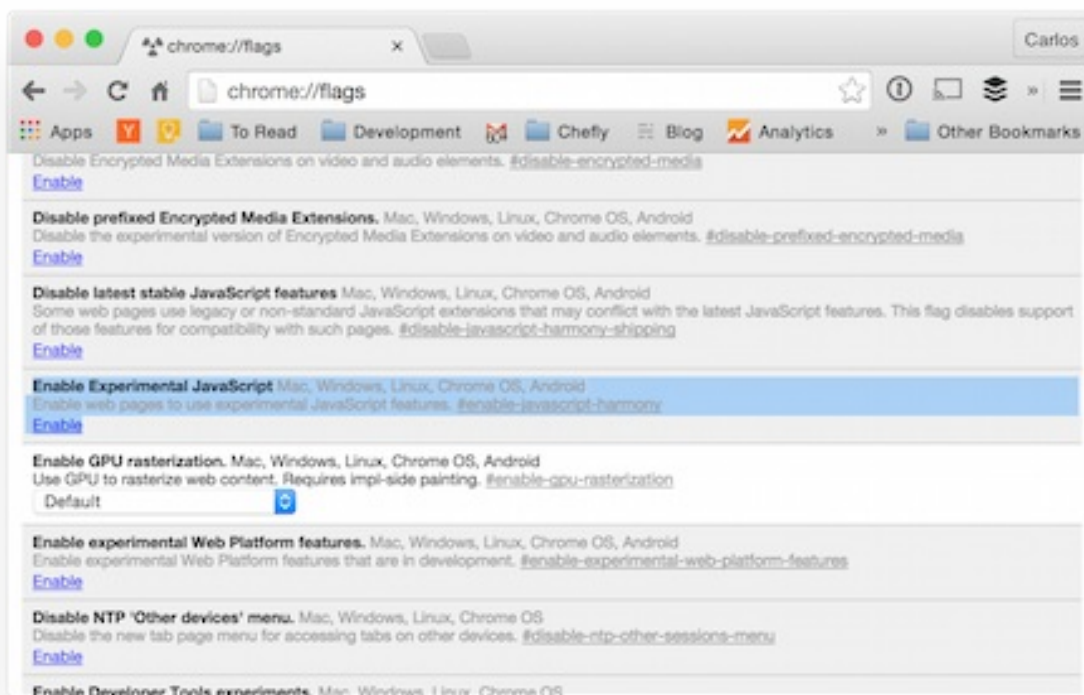
```
import {caminar, correr} from './modulo'
```

ES6 trae muchas más novedades, pero se escapan del ámbito de este libro. Poco a poco los navegadores web van poco a poco incluyendo nuevas funcionalidades, pero sin lugar a dudas, la mejor manera de utilizarlo hoy en día es con el *transpilador* **Babel** para hacer uso de todas las novedades que nos proporciona el nuevo estándar.

Cómo utilizarlo hoy

¿Cómo podemos empezar a utilizarlo hoy en día?. Lo primero más recomendable es que te instales la última versión de Chrome, que es uno de los navegadores que está implementando las nuevas features de ES6 más rápidamente. Te aconsejo incluso que instales Chrome Canary, que es la versión de Chrome que prueba funcionalidades antes de lanzarlas en el Chrome original.

Para probar directamente código ES6 en la consola de tu navegador. Escribe en la barra de direcciones `chrome://flags` y tendrás una página como ésta:



Y activar el flag **Enable Experimental JavaScript**.

Esto te permitirá probar algunas features pero no todas porque aún están en desarrollo.

La otra opción y más utilizada hasta el momento es utilizar un **transpiler** como es el caso de **Babel** (Anteriormente conocido como *6to5*).

Para poder utilizarlo necesitas tener instalado *Node.js* en tu equipo. Lo puedes descargar desde su [web oficial](#) Después con el gestor de paquete de node, *npm* podemos instalarlo de forma global desde la terminal con el comando:

```
$ npm install -g babel-cli
```


Y utilizarlo es muy simple. Si tenemos un archivo con código en ES6 y queremos transformarlo a ES5, sencillamente con el siguiente comando generaremos un fichero `.js` con las instrucciones en ES5:

```
babel archivo_es6.js -o archivo_final.js
```

`archivo_final.js` tendrá el código en el estándar que entienden la mayoría de los navegadores.

Cuidado con la versión 6 de Babel Con la actualización del transpilador, hay que realizar una serie de modificaciones para conseguir traducir el código.

Babel v6 fue modificado y dividido en varios módulos, si queremos utilizar todo lo que trae ES6 tenemos que instalar en nuestro proyecto el módulo `babel-preset-es2015` e incluirlo en un fichero `.babelrc` o dentro del `package.json` con las dependencias del proyecto.

Por ejemplo, un `package.json` de un proyecto tipo sería así:


```
{
  "name": "proyecto",
  "description": "Ejemplo de proyecto ES6",
  "version": "1.0.0",
  "scripts": {
    "build": "babel src -d build"
  },
  "devDependencies": {
    "babel-cli": "^6.0.0",
    "babel-preset-es2015"
  },
  "babel": {
    "presets": [
      "es2015"
    ]
  }
}
```

En este fichero hemos definido también un script de `npm`, con lo que cuando corramos el comando `npm run build` se ejecutará `babel src -d build` que lo hace es correr *Babel* en un directorio llamado `src` y *transpilar* todos los archivos y dejarlo en otra carpeta llamada `build`.

Y para que *Babel* utilice el plugin de `es2015` lo hemos definido en un objeto `babel` dentro del `package.json` dentro del array `presets`, ya que Babel tiene muchos plugins y *presets* más como `react`, ES7 experimental, etc...

Twittea sobre este libro

Si te gustó el libro, no tienes más que enviar un tweet para apoyarlo.

 [#AprendiendoJavaScript](#) con el nuevo ebook de [@carlosazaustre](#).

Descubre lo que otras personas está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter: <https://twitter.com/search?q=#AprendiendoJavascript>