



**actibyti** ] a netmind  
social project

ES6

© 2017, ACTIBYTI PROJECT SLU, Barcelona  
Autor: Ricardo Ahumada



GOBIERNO  
DE ESPAÑA

MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

**red.es**



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



**UNIÓN EUROPEA**

Fondo Social Europeo  
“El FSE invierte en tu futuro”

# ÍNDICE DE CONTENIDOS

1. Introducción
2. Características ES6
3. Destructores
4. Módulos
5. Webpack
6. Promesas

1

# Introducción

# ES6

- JavaScript es un superconjunto del lenguaje de programación ECMAScript.
- ECMAScript forma la base de lenguajes JavaScript, Jscript y ActionScript.
- ECMAScript 6, también conocido como ECMAScript 2015, es la última versión del estándar ECMAScript.
  - <http://www.ecma-international.org/ecma-262/6.0/>
- ES6 es una actualización significativa del lenguaje, y la primera actualización a la lengua desde ES5 se estandarizó en 2009.
- La implementación de estas características en los motores JavaScript principales está en marcha ahora.



# Actualización ES6

## › Incluye las siguientes características

- › arrows
- › classes
- › enhanced object literals
- › template strings
- › destructuring
- › default + rest + spread
- › let + const
- › iterators + for..of
- › generators
- › unicode
- › modules
- › module loaders
- › map + set + weakmap + weakset
- › proxies
- › symbols
- › subclassable built-ins
- › promises
- › math + number + string + array + object APIs
- › binary and octal literals
- › reflect api
- › tail calls

# Compatibilidad ECMAScript6

- Los navegadores actuales no son compatibles con todas las nuevas características de ECMAScript 6.
  - En la pagina web <http://kangax.github.io/compat-table/es6/> y <http://caniuse.com/#search=es6> podemos observar tablas de compatibilidad de ES6, donde se enumeran las características de ES6 y los navegadores que lo soportan y los que no.

		Compilers/polyfills							Desktop browsers									
		95%	56%	71%	48%	59%	18%	5%	11%	93%	96%	96%	94%	97%	97%	97%	97%	
Feature name	▶	Current browser	Traceur	Babel + core-js <sup>[2]</sup>	Closure	TypeScript + core-js	es6-shim	KO 4.14 <sup>[3]</sup>	IE 11	Edge 14	Edge 15	Edge 16 Preview	FF 52 ESR	FF 54 Beta	FF 55 Nightly	CH 59, OP 46 <sup>[1]</sup>	CH 60, OP 47 <sup>[1]</sup>	
		Optimisation	proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	
Syntax	▶	default function parameters	7/7	4/7	4/7	4/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	6/7	7/7	7/7	7/7	
rest parameters	▶	5/5	4/5	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
spread (...) operator	▶	15/15	15/15	13/15	12/15	4/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	
object literal extensions	▶	6/6	6/6	6/6	4/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	
for...of loops	▶	9/9	9/9	9/9	6/9	3/9	0/9	0/9	0/9	7/9	9/9	9/9	7/9	9/9	9/9	9/9	9/9	
octal and binary literals	▶	4/4	2/4	4/4	4/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	
template literals	▶	5/5	4/5	4/5	3/5	3/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
RegExps 'y' and 'u' flags	▶	5/5	3/5	3/5	0/5	0/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	
destructuring declarations	▶	22/22	20/22	21/22	19/22	15/22	0/22	0/22	0/22	21/22	22/22	22/22	21/22	22/22	22/22	22/22	22/22	
destructuring assignment	▶	24/24	23/24	24/24	17/24	19/24	0/24	0/24	0/24	23/24	24/24	24/24	23/24	24/24	24/24	24/24	24/24	

# Transpilación

- Por tanto, necesita utilizar un compilador (transpiler) para transformar su código ECMAScript 6 en código compatible con ECMAScript 5.
- Aunque hay otras opciones, Babel se ha convertido en el estándar de facto para compilar aplicaciones ECMAScript 6 a una versión de ECMAScript que puede ejecutarse en navegadores actuales.
  - <https://babeljs.io/>
- Babel también puede compilar otras versiones de ECMAScript y JSX de React.

# Categorización de ES6

- Todas las características nuevas de ES6 podemos clasificarlas en 7 categorías:
  - Variables y parámetros.
  - Clases.
  - Proxies.
  - Construir en objetos y características basadas en los prototipos.
  - Promesas.
  - Reflejar API
  - Módulos
  - Iteradores y generadores.



# Instalando Babel

# Crea el proyecto



- Crea un directorio de proyecto y genera un archivo **js/main.js** con código ES6

```
let texto="Hola mundo";
console.log("ES6",texto)
```

- Genera un **index.html** con una estructura base HTML5

# Configura el proyecto



1. Abre una ventana de consola en el directorio del proyecto y añade un package.json

```
npm init
```

2. Añade **babel-cli** y **babel-core** al proyecto

```
npm install babel-cli babel-core --save-dev
```

3. Instala **ECMAScript 2015 preset**

```
npm install babel-preset-es2015 --save-dev
```

- › En Babel 6, cada transformador es un plugin que se puede instalar por separado.
- › Un **preset** es un grupo de complementos relacionados.
- › Usando un preset evita tener que instalar y actualizar docenas de plugins individualmente.

# Configura el proyecto



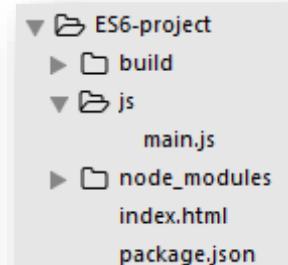
4. Abre package.json para editar.

- En la sección de scripts, elimina el script **test** y añade dos nuevos scripts:
- un script llamado **babel** que compila main.js a una versión de ECMAScript que pueda ejecutarse en navegadores actuales
  - un script llamado **start** que inicie el servidor web local.

```
"scripts": {  
  "babel": "babel --presets es2015 js/main.js -o build/main.bundle.js",  
  "start": "lite-server"  
},
```

5. En el directorio del proyecto crea el directorio **build**, que contendrá la versión compilada

- La estructura del proyecto debe quedar así.



# Compila y ejecuta el proyecto



- En la línea de comandos, en el directorio de proyecto y escriba el siguiente comando para ejecutar el script babel y compilar main.js:

```
npm run babel
```

- Se habrá generado una versión compilada de main.js en build/main.bundle.js
- Abre index.html y editalo para que cargue el script compilado **build/main.bundle.js**

```
<script src="build/main.bundle.js"></script>
```

- Lanza el servidor

```
npm start
```

2

## Características ES6

# let

- **let** Permite definir variables de ámbito bloque (intervalo de paréntesis) en JavaScript.
- Inicialmente, JavaScript solo soportaba el ámbito de funciones y las variables de ámbito global.
- Ejemplo:

```
if(true){  
    let x = 12;  
    alert(x); //alert's 12  
}  
  
alert(x); //x is undefined here
```

- La palabra clave “let” limita la accesibilidad de las variables en un bloque, una sentencia o una expresión.

## const

- **const** permite definir variables de solo lectura (una constante).
- Las variables creadas con **const** tienen un ámbito de bloque.
- Si intentamos redeclarar una variable const en el mismo ámbito nos indicará un error.

```
const x = 12;
```

# Const - Ejemplo

```
const x = 12;
```

*//an constant 'x' is already available in this scope therefore the below line throws an error when you are try to create a new x variable.*

```
const x = 13;
```

```
if(true) {
```

*//an constant 'x' is available in this scope but not defined in this scope therefore the below line will not throw error instead define a new "x" inside this scope.*

```
    const x = 13;
```

*//here 'y' is available inside this scope not outside this scope*

```
    const y = 11;
```

```
}
```

*//here creating a new 'y' will not throw an error because no other 'y' is available in this scope(i.e., global scope)*

```
    const y = 12;
```

# Funciones con retorno de valores múltiples

- ES6 proporciona una matriz como sintaxis para asignar múltiples variables de valores de índices de matriz.
- También permite ignorar algunos índices de matriz.
- Ejemplo:

```
function function_name() {  
    return [1, 6, 7, 4, 8, 0]; //here we are storing variables in an array and returning the array  
}  
  
var q, w, e, r, t, y;  
  
//Here we are using ES6's array destructuring feature to assign the returned values to variables.  
//Here we are ignoring 2 and 4 array indexes  
[q, w, , r, , y] = function_name();  
  
alert(y); //y is 0
```

# Argumentos predeterminados de una función

- ES6 nos ofrece una nueva sintaxis que se puede utilizar para definir los valores por defecto para parámetros de la función:

```
function myFunction(x = 1, y = 2, z = 3){  
    console.log(x, y, z); // Output "6 7 3"  
}  
  
myFunction(6,7);
```

- Además, ***undefined*** es considerado como falta de argumento. Veamos un ejemplo donde podemos demostrar esto:

```
function myFunction(x = 1, y = 2, z = 3)  
{  
    console.log(x, y, z); // Output "1 7 9"  
}  
  
myFunction(undefined,7,9);
```

# Argumentos predeterminados de una función

- Los valores predeterminados también pueden ser expresiones.  
Veamos un ejemplo:

```
function myFunction(x = 1, y = 2, z = 3 + 5)
{
    console.log(x, y, z); // Output "6 7 8"
}
myFunction(6,7)
```

## Operador “...”

- ES6 introdujo el operador `...` que también es llamado operador de difusión (**spread**). Cuando el operador “...”
- Se aplica en una matriz y expande la matriz en varias variables en forma sintáctica.
- Cuando se aplica a un argumento de función, hace que se comporte como una matriz de argumentos.
- Podemos utilizar el operador de difusión para tomar un numero indefinido de argumentos.

# Ejemplo Operador ...

```
//args variable is an array holding the passed function arguments
function function_one(...args)
{
    console.log(args);
    console.log(args.length);
}

function_one(1, 4);
function_one(1, 4, 7);
function_one(1, 4, 7, 0);

function function_two(a, b, ...args)
{
    console.log(args);
    console.log(args.length);
}

//args holds only 7 and 9
function_two(1, 5, 7, 9);
```

## Operador “...”

- Antes de que ES6 introdujera “...” los desarrolladores estaban acostumbrados a usar `array.prototype.slice.call` para recuperar los argumentos adicionales pasados a la función, como muestra el siguiente código:

```
//args variable is an array holding the function arguments
function function_one(){
    var args = Array.prototype.slice.call(arguments, function_one.length);

    console.log(args);
    console.log(args.length);
}

function_one(1, 4);
function_one(1, 4, 7);
function_one(1, 4, 7, 0);
function function_two(a, b){
    var args = Array.prototype.slice.call(arguments, function_two.length);

    console.log(args);
    console.log(args.length);
}

//"args" holds only 7 and 9
function_two(1, 5, 7, 9);
```

## Operador “...”

- Si aplicamos “...” a una matriz, se expande en la sintaxis de multiples variables.
- Ejemplo:

```
function function_name(a, b){  
    console.log(a+b);  
}  
  
var array = [1, 4];  
  
function_name(...array); //is equal to function_name(1, 4)
```

# Cadenas multilinea

- Una cadena multilinea (o cadena de bloque) es una cadena que se distribuye en varias líneas y que también conserva caracteres de salto de línea.
- En JavaScript: básicamente existen dos métodos para lograrlo.

## Método 1: utilizando el carácter “\”.

- Permite extender la cadena sobre varias líneas, pero no conserva los caracteres de línea nueva.

```
var string = "This is first line \n\  
This is second line \n\  
This is third line \n\  
";  
console.log(string);
```

# Cadenas multilinea

**Método 2:** usar ECMAScript Template String, usando el carácter `.

```
var name = "narayan";
var x = `
My Name is ${name}.
My Profession is web development.
`;
console.log(x);
```

- ECMAScript Template String también admite la función de cadenas como se muestra anteriormente.

# Características de ES6

**Método 3:** este método es básicamente una forma **hackeada** para conseguir esta característica.

- Este método convierte los comentarios JavaScript en una cadena multilinea preservando los nuevos caracteres de línea.
- Ejemplo:

```
var myString = (function () {/*
    I am narayan
    I am a web developer
*/}).toString().match(/[^\n]*\n*/)[1];

console.log(myString);
```

# class

- ECMA Script introdujo la palabra clave **class** para definir clases en JavaScript.
- Las clases ES6 son una evolución sobre el patrón OO basado en prototipo.
- Tener una única forma declarativa conveniente hace que los patrones de clase sean más fáciles de usar, y fomenta la interoperabilidad.
- Las clases soportan **herencias** basadas en prototipos, llamadas **super**, **instancias** y **métodos estáticos** y **constructores**.

# Ejemplo definición de clase

```
class Student{
    //constructor of the class
    constructor(name, age) {
        //this points to the current object
        this.name = name;

        this._age = age;
    }

    //member function
    getName() {
        return this.name;
    }

    setName(name) {
        this.name = name;
    }

    set age(value) {
        this._age = value;
    }

    get age() {
        return this._age;
    }
}
```

# Ejemplo de herencia de clase

```
//class person inherits student class
class Person extends Student{
    constructor(name, age, citizen) {
        //this points to the person class
        this.citizen = citizen;

        //call constructor of super class. "super" is an pointer to the super class object
        super(name, age);
    }

    getCitizen() {
        return this.citizen;
    }

    //overriding
    getName() {

        //we are calling the super class getName function
        return super.getName();
    }
}
```

# Instanciación de objetos

- Asimismo, con las clases definidas podemos instanciar objetos:

```
//instance of student class
var stud = new Student("Narayan", 21);

//instance of person class
var p = new Person("Narayan Prusty", 21, "India");

stud.age = 12; //executes setter
console.log(stud.age); //executes getter
```

## Función arrow “=>”

- ES6 proporciona una nueva manera de crear funciones que solo contienen una línea de declaración.
- Este nuevo tipo de función es llamada **lambda** o **flecha (arrow)** y se representa mediante el operador `=>`

```
//sum is the function name
//x and y are function parameters
var sum = (x, y) => x + y;

console.log(sum(2, 900)); //902
```

- Aquí `(x, y) => x + y` devuelve un objeto función JavaScript normal equivalente a `function(x, y){return x+ y;}`
- Las funciones arrow siempre devuelven el valor de la sentencia cuando se ejecutan.
  - En el ejemplo se devuelve el resultado de  $x+y$

## Función arrow “=>”

- Tambien podemos escribir varias sentencias en una función de flecha, pero las funciones arrow se utilizan principalmente en la sustitución de las funciones de los estados individuales.
- Ejemplo:

```
var sum = (x, y) => {  
    x = x + 10;  
    y = y + 10;  
    return x + y;  
}  
  
console.log(sum(10, 10)); //40
```

# Función arrow “=>”

- Debido a que la función arrow devuelve un objeto de función, puede ser usado dondequiero que usemos un objeto de función JavaScript regular.
- Por ejemplo, pueden utilizarse como devolución de llamada.

```
function sum(p, q){  
    console.log(p() + q()); //87  
}  
  
sum(a => 20 + 10, b => 1 + 56); //here we are passing two function objects
```

## Función arrow “=>”

- Una de las características más importante de la función arrow es el puntero “**this**”, asincrónicamente ejecutado, apunta al ámbito dentro del cual se paso la llamada.
- En una función regular **this** apunta a ámbito global cuando se ejecuta de forma asíncrona.

```
window.age = 12;

function Person(){
    this.age = 34;

    setTimeout(() => {
        console.log(this.age); //34
    }, 1000);

    setTimeout(function(){
        console.log(this.age); //12
    }, 1000);
}

var p = new Person();
```

## for of

- **for of** itera sobre los valores de los elementos de una colección.
- Una colección puede ser una matriz, un conjunto, una lista, un objeto de colección personalizado, etc.
- Un **iterador** es un constructor que nos permite visitar o recorrer todos los elementos de una colección

```
var array = [ 1 , 3 , 5 , 7 , 9 ] ;  
  
// 'i' referencia a los valores de los índices de matriz  
for(var i of array){  
    console.log(i); //1, 3, 5, 7, 9  
}
```

- Internamente el bucle *for of* utiliza el metodo **@@iterator** de un objeto de colección, es decir, para que un objeto de recopilación sea iterable utilizando *for of* debe tener propiedad con una clave **symbol.iterator**

# Iteración de un objeto de colección personalizado

- Debemos implementar la propiedad **symbol.iterator** en una colección personalizada.
- **symbol.iterator** devuelve un objeto iterador, es decir, un objeto con la propiedad **next()**.
- *next()* es invocado por **for of** hasta que *next()* devuelve **{value: undefined, done: true}**.
- Para continuar el bucle y devolver un elemento de colección *next()* tiene que devolver **{value: element\_value, done: false}**.

# Iteración de un objeto de colección personalizado

```
var custom_collection = {
  elements: [1, 4, 6, 9],
  size : 3,
  pointer :0,
  [Symbol.iterator]: function(){
    var e = this.elements;
    var s = this.size;
    var p = this.pointer;
    return{
      next: function() {
        if(p > s)
        {
          return { value: undefined, done: true };
        }
        else
        {
          p++;
          return { value: e[p - 1], done: false };
        }
      },
    };
  }
}
```

```
for(var i of custom_collection)
{
  console.log(i); //1, 4, 6, 9
}
```

# Generators

- La palabra clave **yield** y la sintaxis de **function()** de javascript, forman juntos un Generator.
- Un Generator proporciona una nueva manera de realizar funciones para devolver una colección y; también una nueva forma de realizar bucles o iteraciones a través de los elementos de la colección devuelta.
- Un ejemplo del código sin JavaScript Generator es el siguiente:

```
function collection_name(){  
    return [1, 3, 5, 7];  
}  
  
var collection = collection_name();  
  
for(var iii = 0; iii < collection.length; iii++) {  
    console.log(collection[iii]);  
}
```

# Generators

- Podemos hacer exactamente lo mismo usando Generators, de esta manera:

```
function* collection_name() {  
    yield 1;  
    yield 3;  
    yield 5;  
    yield 7;  
}  
  
for(var iii of collection_name()) {  
    console.log(iii);  
}
```

- Internamente JavaScript crea un objeto **Symbol.iterator** con propiedad de los valores producidos (yielded).

# El objeto Set

- Es una colección de claves únicas (conjuntos). Las claves son referencias de objetos o tipos primitivos.
- Los arrays pueden almacenar valores duplicados pero los Conjuntos no almacenan claves duplicadas, esto es lo que la hace diferente de los arrays.
- A continuación se muestra un ejemplo de como crear un objeto de tipo Set, agregar claves, eliminarlas, encontrar el tamaño, entre otros casos.

```
//create a set
var set = new Set();

//add three keys to the set
set.add({x: 12});
set.add(44);
set.add("text");

//check if a provided key is present
console.log(set.has("text"));
```

# El objeto Set ...

```
//delete a key
set.delete(44);

//loop through the keys in an set
for(var i of set){
    console.log(i);
}

//create a set from array values
var set_1 = new Set([1, 2, 3, 4, 5]);

//size of set
console.log(set_1.size); //5

//create a clone of another set
var set_2 = new Set(set.values());
```

# Set y WeakSet

- Set y WeakSet permiten almacenar conjuntos de claves únicas, sin embargo, existen diferencias entre ellos
  - Ambos se comportan de manera distinta cuando un objeto referenciado por sus claves se elimina.

```
var set = new Set();
var weakset = new WeakSet();

(function(){
    var a = {x: 12};
    var b = {y: 12};

    set.add(a);
    weakset.add(b);
})()
```

- La función invocada, si se ejecuta, no hay manera que podamos hacer referencia a {x: 12} y {y: 12}, el recolector de basura sigue adelante y elimina la clave b del puntero WeakSet y también elimina {y: 12} de la memoria.
- En el caso de Set, puede causar mas basura en la memoria.
- **Podemos decir que Set es un indicador fuerte mientras que WeakSet es un indicador mas débil.**

# Características de ES6

- Las claves de WeakSet no pueden ser de tipo primitivos, tampoco pueden ser creados por una matriz o un conjunto.
- WeakSet no proporciona ningún método o funciones para trabajar con todo el conjunto de claves.

```
var set = new Set([1,2,3,4]);  
  
//cannot be created from array or another set  
var weakset = new WeakSet();  
weakset.add({a: 1}); //object reference must
```

```
var set = new Set([1,2,3,4]);  
  
var weakset = new WeakSet();  
weakset.add({a: 1}); //object reference must  
  
console.log(set.size); //4  
console.log(weakset.size); //undefined  
  
for(var i of set){  
    console.log(i); //1,2,3,4 }  
  
for(var i of weakset) {  
    console.log(i); }  
  
set.clear();  
weakset.clear(); //This works
```

# Objeto Map

- Es un conjunto de claves únicas y sus valores correspondientes. Las claves y los valores pueden hacer referencias a objetos o tipos primitivos.
- Los Arrays 2D pueden almacenar valores duplicados, pero los maps no almacenan claves duplicadas, lo cual lo hace diferente a los arrays 2D.
- Un objeto Set en JavaScript puede almacenar claves, pero maps puede almacenar pares de claves y valores.

```
//create a map
var map = new Map();

//add three keys & values to the map
map.set({x: 12}, 12);
```

## Objeto Map ...

```
//same key is overwritten
map.set(44, 13);
map.set(44, 12);

//check if a provided key is present
console.log(map.has(44)); //true

//retrieve key
console.log(map.get(44)); //12

//delete a key
map.delete(44);

//loop through the keys in an map
for(var i of map)
{
  console.log(i);
}
```

```
//delete all keys
map.clear();

//create a map from arrays
var map_1 = new Map([[1, 2], [4, 5]]);

//size of map
console.log(map_1.size); //2
```

3

## Destructores

# Destructores

- ECMAScript 6 ha introducido una nueva sintaxis que hace que sea sencillo crear objetos en función de variables, por el contrario la nueva sintaxis objeto y matriz de desestructuración hace que sea fácil crear variables basadas en objetos y matrices.
- En el siguiente ejemplo, devolveremos diferentes valores para la función CalculateMonthlyPayment : Pago mensual, tasa mensual y otros parámetros de hipoteca.
- Lo primero, es abrir **js/main.js** en el editor de código.
- Modificaremos la instrucción de retorno de la función CalculateMonthlyPayment :

# Destructores

```
return {principal, years, rate, monthlyPayment, monthlyRate};
```

- El código anterior es una manera abreviada de la siguiente sintaxis realizada por ECMAScript 5:

```
return { principal: principal,  
        years: years,  
        rate: rate,  
        monthlyPayment: monthlyPayment,  
        monthlyRate: monthlyRate  
    };
```

# Destructores

- El segundo paso, es abrir index.html y añadir el bloque `<h3>` de abajo para mostrar la tasa mensual debajo del pago mensual, como se muestra a continuación:

```
<h2>Monthly Payment: <span id="monthlyPayment" class="currency"></span></h2>
<h3>Monthly Rate: <span id="monthlyRate"></span></h3>
```

- Luego, abrimos main.js en el controlador de eventos calcBtn, y modificamos la llamada a calculateMonthlyPaymentla, de la siguiente manera:

```
let {monthlyPayment, monthlyRate} = calculateMonthlyPayment(principal, years, rate);
```

# Destructores

- El código anterior es una manera abreviada del siguiente código de ECMAScript 5:

```
var mortgage = calculateMonthlyPayment(principal, years,  
rate);  
var monthlyPayment = mortgage.monthlyPayment;  
var monthlyRate = mortgage.monthlyRate;
```

- En la última línea de calcBtn añada la siguiente línea de código para mostrar la tasa mensual luego del pago mensual, de la siguiente manera:

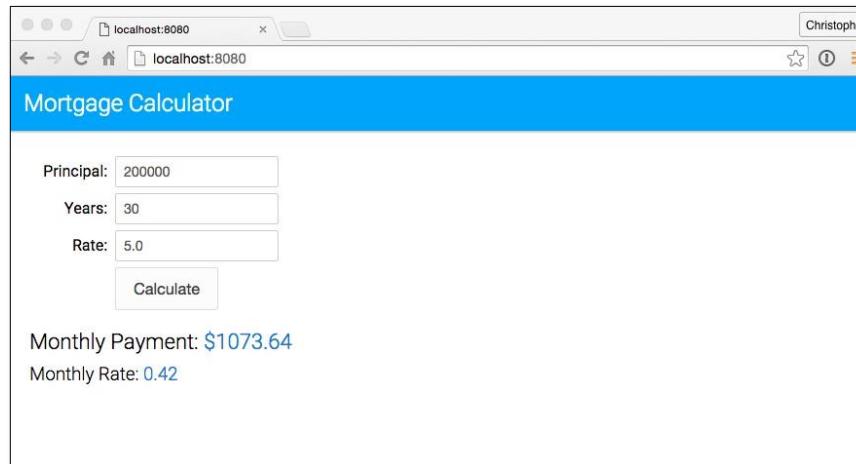
```
document.getElementById("monthlyRate").innerHTML = (monthlyRate * 100).toFixed(2);
```

# Destructores

- El último paso es ingresar a la línea de comandos y escribir el siguiente código para generar la aplicación:

```
npm run babel
```

- Ingrese al navegador, coloque la siguiente URL **http://localhost:8080** y haga clic en el botón **calcular**, de la siguiente manera:



4

## Módulos

# Creando Módulos JavaScript

- Los módulos de JavaScript son una forma de empaquetamiento relacionado con el código JavaScript en su propio ámbito que puede ser consumido por otro programa JavaScript.
- En este contenido, veremos las diferentes maneras de crear módulos JavaScript.

## Immediately-Invoked Function Expression (IIFE)

- Es una función anónima que se invoca a sí misma.
- La creación de módulos utilizando IIFE es la forma mas antigua de crear módulos en JavaScript.
- En el siguiente ejemplo se muestra como crear un módulo JavaScript utilizando IIFE.

# Immediately-Invoked Function Expression (IIFE)

```
(function(window){  
  
    var sum = function(x, y){  
        return x + y;  
    }  
  
    var sub = function(x, y){  
        return x - y;  
    }  
  
    var math = {  
        findSum: function(a, b){  
            return sum(a,b);  
        },  
  
        findSub: function(a, b){  
            return sub(a, b);  
        }  
    }  
  
    window.math = math;  
  
})(window);
```

- Usamos el módulo

```
console.log(math.findSum(1, 2)); //3  
console.log(math.findSub(1, 2)); //-1
```

- Los objetos de función sum y sub permanecen en la memoria, pero no hay forma en que el programa que utiliza el módulo pueda acceder a ellos, por lo cual se impide cualquier posibilidad de las variables globales.

## CommonJS

- CommonJS es una especificación JavaScript que no es para navegador para crear módulos. Se utiliza principalmente en NodeJS.
- Supongamos que tenemos dos archivos **main.js** y **math.js**, vamos a hacer que **math.js** se comporte como un módulo y luego sea utilizado en el archivo **main.js**.

# CommonJS

## ➤ Archivo **math.js**

```
var sum = function(x, y){  
    return x + y;  
}  
  
var sub = function(x, y){  
    return x - y;  
}  
  
var math = {  
    findSum: function(a, b){  
        return sum(a,b);  
    },  
  
    findSub: function(a, b){  
        return sub(a, b);  
    }  
}  
  
//All the variable we want to expose outside needs to be property of "exports" object.  
exports.math = math;
```

# CommonJS

## ➤ Archivo **main.js**

```
//no file extension required
var math = require("./math").math;
console.log(math.findSum(1, 2)); //3
console.log(math.findSub(1, 2)); //-1
```

- En este ejemplo hemos evitado con éxito cualquier posibilidad de sustituir variables globales y también hemos podido ocultar la implementación.

# Asynchronous Module Definition (AMD)

- Es una especificación para crear módulos para el navegador.
- Está diseñado con limitaciones de navegador, que no soportan nativamente AMD por lo cual necesitan de librería de aplicación para crear e importar módulos de AMD, una de ellas es **RequireJS**.
- Veamos un ejemplo sobre la utilización de RequireJS para crear e importar un módulo, supongamos que tenemos un archivo **index.js**, que posee el código JS de nuestro sitio web, y queremos crear un módulo llamado “math”, el cual importaremos en el archivo **index.js**
- Código para **index.html**

# Asynchronous Module Definition (AMD)

## ➤ Código del archivo **index.html**

```
<!doctype html>
<html>
  <head></head>
  <body>
    <!-- Load RequireJS library and then provide relative path to our website's
        JS file. File extension not required. -->
    <script type="text/javascript"
      src="http://requirejs.org/docs/release/2.1.16/minified/require.js" data-
      main="index"></script>
  </body>
</html>
```

## ➤ Proporcionaremos la URL relativa del archivo **index.js** en el atributo data-main.

# Creando Módulos JavaScript

## ➤ Código del archivo **index.js**

```
//list of modules required
require(["math"], function(math){
    //main program
    console.log(math.findSum(1, 2)); //3
    console.log(math.findSub(1, 2)); //-1
})
```

- Pedimos a RequireJS cargar el módulo math antes de que inicie el programa principal.

# Creando Módulos JavaScript

- Código del archivo **math.js**
- Ahora cualquier sitio web que utilice RequireJS puede importar el **math.js**

```
define(function(){  
  
    var sum = function(x, y){  
        return x + y;    }  
  
    var sub = function(x, y){  
        return x - y;    }  
  
    var math = {  
        findSum: function(a, b){  
            return sum(a,b);  
        },  
  
        findSub: function(a, b){  
            return sub(a, b);  
        }  
    }  
  
    return math;  
});
```

# Módulos ES6

- Debido a la gran utilización de módulos de JavaScript es el momento de que los navegadores y motores del lado del servidor puedan soportar de forma nativa alguna forma de sistema de módulo común, por tal motivo surgieron los módulos ES6, los cuales son compatibles en todas partes.
- En ES6 cada módulo se representa por un archivo de clase que se puede exporta mediante la palabra clave **export**.

```
export class Math {  
    ....  
}
```

# Módulos ES6

- Código del módulo **Math** en el archivo **math.js**

```
export class Math {  
  
    constructor()  
    {  
        this.sum = function(x, y){  
            return x + y;  
        }  
  
        this.sub = function(x, y){  
            return x - y;  
        }  
    }  
  
    findSum(a, b)  
    {  
        return this.sum(a, b);  
    }  
  
    findSub(a, b)  
    {  
        return this.sub(a, b);  
    }  
}
```

## Módulos ES6

- Para importar y usar el módulo, usaremos **import** indicando el archivo fuente (sin extensión)

```
import {Math} from 'math';

var math = new Math();

console.log(math.findSum(1, 2)); //3
console.log(math.findSub(1, 2)); //-1
```

5

# Webpack

# Webpack

- Al momento de compilar una aplicación modular ECMAScript 6 a ECMAScript 5, el compilador se basa en una biblioteca de terceros para implementar módulos en ECMAScript 5.
- Webpack (<http://webpack.github.io/>) y Browserify (<http://browserify.org/>) son dos opciones populares, y Babel soporta ambas (y otras).
- Primero que debemos hacer es configurar Webpack. En la línea de comandos, instalaremos los módulos babel-loader y Webpack:

```
npm install babel-loader webpack --save-dev
```

# Webpack

- Luego, editaremos **package.json** y añadiremos un script webpack:

```
"scripts": {  
  "babel": "babel --presets es2015 js/main.js -o build/main.bundle.js",  
  "start": "http-server",  
  "webpack": "webpack"  
},
```

# Webpack

- Crearemos un nuevo archivo **webpack.config.js** con la configuración de webpack:

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  entry: './js/main.js',
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'main.bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015']
        }
      }
    ]
  },
  stats: {
    colors: true
  },
  devtool: 'source-map'
};
```

# Webpack

- Para compilar el módulo usaremos la siguiente orden en línea de comandos:

```
npm run webpack
```

- Por último abriremos el navegador en la dirección **http://localhost:8080** y probar la funcionalidad.

6

# Promesas

# Promesas

- Las promesas han reemplazado las funciones de devolución de llamada y es el estilo de programación preferido para el manejo de llamadas asíncrónicas.
- Una promesa es el titular de un resultado (o un error) que estará disponible en el futuro (cuando la llamada asíncrona vuelva).
- Las promesas han estado disponibles en JavaScript a través de bibliotecas de terceros (por ejemplo, jQuery y q).
- ECMAScript 6 agrega soporte incorporado para las promesas de JavaScript.



## Usando promesas

- Veamos un ejemplo simple, llamado retafinder que devuelve una lista de los tipos hipotecarios disponibles.

# Promesas

- Creamos un archivo con el nombre **ratefinder.html**, con el siguiente código:
- Creamos un archivo **ratefinder.js**:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <table id="rates"></table>
  <script src="build/ratefinder.bundle.js"></script>
</body>
</html>
```

```
let url = "rates.json";

fetch(url)
  .then(response => response.json())
  .then(rates => {
    let html = "";
    rates.forEach(rate => html +=
`<tr><td>${rate.name}</td><td>${rate.years}</td><td>${rate.rate}%</td></tr>`);
    document.getElementById("rates").innerHTML = html;
  })
  .catch(e => console.log(e));
```

# Promesas



- Editamos **webpack.config.js**; en **module.exports**, modificar los elementos de entrada y salida de la siguiente manera:

```
entry: {  
    app: './js/main.js',  
    ratefinder: './js/ratefinder.js'  
},  
output: {  
    path: path.resolve(__dirname, 'build'),  
    filename: '[name].bundle.js'  
},
```

- Generamos la aplicación con webpack:

```
npm run webpack
```

- Abrimos el navegador, con la siguiente url  
**http://localhost:8080/ratefinder.html**.

# Promesas



- El segundo paso, es crear una promesa.
- Lo primero a realizar es crear un nuevo archivo con el nombre de **rate-service-mock.js** en el directorio js.
- En **tasa-service-mocj.js.js**, definimos una variable **rates**, con algunos datos de muestra como se ve a continuación:

```
let rates = [  
    {  
        "name": "30 years fixed",  
        "rate": "13",  
        "years": "30"  
    },  
    {  
        "name": "20 years fixed",  
        "rate": "2.8",  
        "years": "20"  
    }  
];
```

# Promesas



- Definimos una función findAll():

```
export let findAll = () => new Promise((resolve, reject) => {
  if (rates) {
    resolve(rates);
  } else {
    reject("No rates");
  }
});
```

- Modificamos **ratefinder.js**:

```
import * as service from './rate-service-mock';
service.findAll()
  .then(rates => {
    let html = '';
    rates.forEach(rate => html +=
`|  |  |  |
| --- | --- | --- |
| ${rate.name} | ${rate.years} | ${rate.rate}% |
`);
    document.getElementById("rates").innerHTML = html;
  })
  .catch(e => console.log(e));
```

# Promesas



- Generamos la aplicación:

```
npm run webpack
```

- Y Abriremos un navegador con la siguiente url **http://localhost:8080 / ratefinder.html**.



## BananaTube: Decisión en equipo

- Discute en equipo las ventajas de ES6 sobre Javascript
- Sería una tecnologías por la que apostar?
- Tomad nota de las fortalezas y debilidades que hayas observado, así como, la dificultad de incorporarlo en el proyecto.



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123  
08018 Barcelona  
Tel. 93 304.17.20  
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7  
28020 Madrid  
Tel. 91 442.77.03  
Fax. 91 442.77.07

[www.netmind.es](http://www.netmind.es)



GOBIERNO  
DE ESPAÑA

MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

**red.es**



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



Agenda Digital para España



**UNIÓN EUROPEA**

Fondo Social Europeo  
*"El FSE invierte en tu futuro"*