ES6

Introducción a ES6

- ECMAScript v6 es el estándar que sigue Javascript desde Junio de 2015.
- Está soportado parcialmente por la mayoría de navegadores, aunque todavía no hay ninguno que lo soporte por completo.
- ES6 es totalmente compatible hacia atrás con versiones anteriores: **Nuestro código actual funcionará perfectamente en navegadores posteriores.**
- La versión 7 ya se encuentra en una etapa avanzada de desarrollo. Promete continuar con los cambios expuestos en la versión 6.
- Necesitaremos algún tipo de traductor para interpretar el código Javascript escrito con ES6.

ES6

- Alguna de las características más importantes agregadas en esta nueva versión son las siguientes:

- **Clases, arrow functions, template strings, herencia, constantes, módulos...**

ES6 - Constantes

- Las constantes presentan un valor dentro del ámbito donde queda definido.
- La característica más reseñable es que dicho valor no puede cambiarse a través de la reasignación.
- Tampoco podemos redeclarar este tipo de elementos.
- Para poder utilizar una constante debemos especificar su valor en la misma sentencia en la que se declara.
- La declaración de una constante crea una referencia de sólo lectura, el identificador que usemos no puede ser reasignado.
- Si el valor contenido dentro de la constante es un objeto, éste sí puede ser alterado.

ES6 - Constantes

- Usamos la palabra reservada **const** para definir nuestras constantes.
- **Sintaxis:**

const *varname1 = value1 [, varname2 = value2 [, varname3 = value3 [, ... [, varnameN = valueN]]]]*;

- Ejemplo:

```
const a = 32
console.log("El valor de a es " + a)
```

ES6 - Constantes

- Podemos declarar las constantes en mayúsculas o minúsculas.
- Por convención se suelen declarar siempre en mayúsculas.

```
const NOMBRE = 'Antonio'

// Error - Assignment to constant variable.
NOMBRE = 'Rosa'
// Error - Identifier 'NOMBRE' has already been declared
const NOMBRE = 'Laura'
// Error - Identifier 'NOMBRE' has already been declared
var NOMBRE = 'Ramon'
```

ES6 - Constantes

- Del mismo modo, podemos trabajar con objetos de tipo constante.
- Podemos modificar el valor de las propiedades del objeto, siempre y cuando no cambiemos la propia definición del mismo.

```
const PERSONA = {
    'nombre': 'Pedro'
}
// Mi nombre es Pedro
console.log("Mi nombre es " + PERSONA.nombre)
// Error - TypeError: Assignment to constant variable.
PERSONA = {
    'name': 'Manuel'
}
PERSONA.nombre = 'Raquel'
// Mi nombre es Raquel
console.log("Mi nombre es " + PERSONA.nombre)
```

ES6 - Constantes

- Lo mismo se puede aplicar en el caso de los Arrays

```
const COLORES = []
COLORES.push('rojo')

// [ 'rojo' ]
console.log(COLORES)

// Error - TypeError: Assignment to constant variable.
COLORES = ['verde']
```

ES6 - Ámbito

- Cuando hablamos de ámbito hacemos referencia a la determinación del alcance de las variables.
- Este ámbito determina qué variables tenemos disponibles en cada bloque de nuestro código.
- En Javascript, objetos y funciones son también variables, por lo que también debemos tener en cuenta qué elementos tenemos al alcance dependiendo del ámbito donde nos encontramos.
- Disponemos de dos ámbitos que debemos conocer a la hora de desarrollar nuestras aplicaciones: **ámbito de función** y **ámbito de bloque**

ES6 - Ámbito

ÁMBITO DE FUNCIÓN

- Las variables que definimos dentro de una función sólo son accesibles desde dentro de la función en sí.
- Lo vemos reflejado en el ejemplo siguiente.

ES6 - Ámbito

```
var x = 'Fuera de la función'

function prueba(){
   // y está definida en el ámbito local
   var y = 'Dentro de la función'

   // x está definida en un ámbito global.
   // podemos acceder a ella dentro de la función
   console.log(x)
}
prueba()

// Error - ReferenceError: y is not defined
console.log(y)
```

ES6 - Ámbito

ÁMBITO DE BLOQUE

- Podemos limitar, de igual manera, el ámbito de una variable, declarándola dentro de un bloque de código (condicional, bucle…)

```
function prueba(){
    const x = 'Dentro de la función'
    if(true){
        const y = 'Dentro del bloque'
        console.log(x) // OK
        console.log(y) // OK
    }
    console.log(x) //OK
    console.log(y) // Error - y is not defined
}

prueba()
```

ES6 - Ámbito

ÁMBITO DE BLOQUE

- En el ejemplo anterior, conseguimos limitar el ámbito de nuestras variables usando el modificador **const**.
- Si queremos asignar un ámbito de función a las diferentes variables definidas, tendríamos que seguir usando **var**.
- Vemos cómo quedaría el ejemplo modificando el ámbito de las variables definidas:

ES6 - Ámbito

ÁMBITO DE BLOQUE

```
function prueba(){
    var x = 'Dentro de la función'
    if(true){
        var y = 'Dentro del bloque'
        console.log(x) // OK
        console.log(y) // OK
    }
    console.log(x) //OK
    console.log(y) // OK
}

prueba()
```

ES6 - Ámbito

ÁMBITO DE BLOQUE

- **¿Por qué observamos este comportamiento?**
- El intérprete de Javascript tiene diferentes fases a la hora de trabajar con nuestros script:
- En la **primera pasada** procesa las variables y la declaración de funciones.
- En la **segunda pasada** procesa el contenido de las funciones y las variables que nos se encuentran declaradas.
- Debido a esta forma de trabajar, las variables procesadas con **var** pueden llegar a convertirse en globales.

ES6 - Ámbito

ÁMBITO DE BLOQUE

- Si queremos evitar este comportamiento, que puede llevarnos a cometer errores, debemos utilizar el nuevo modificador incluído en ES6, **let**.
- Al igual que pasa cuando usamos **const**, definir nuestras variables con **let** nos permite conocer el ámbito donde estamos trabajando con ellas.
- Al contrario que con **const**, trabajando con **let**, sí que podemos modificar el contenido de las variables y reasignar su valor.
- La única acción que no está permitida es la redeclaración de las variables creadas usando let.
- Revisamos el ejemplo anterior usando **let**.

ES6 - Ámbito

ÁMBITO DE BLOQUE

```
function prueba(){
    let x = 'Dentro de la función'
    if(true){
        let y = 'Dentro del bloque'
        console.log(x) // OK
        console.log(y) // OK
    }
    console.log(x) //OK
    console.log(y) // Error - y is not defined
}

prueba()
```

ES6 - Ámbito

ÁMBITO DE BLOQUE

- Un caso interesante a tratar es el la declaración de variables con **let** y **var** dentro de un bucle:

```
for (let i = 0; i < 10; i++) {
   console.log(i);
}

console.log("Fuera del bucle " + i); // Undefined

for (var i = 0; i < 10; i++) {
   console.log(i);
}

console.log("Fuera del bucle " + i); // 10
```

ES6 - Ámbito

ÁMBITO DE BLOQUE

- Existen una serie de normas que nos indican cuándo debemos usar **var, let y const**:
  - Si la variable no va a ser reasignada, debemos usar **const**.
  - Usaremos **let** si la variable será reasignada más adelante (contador de un bucle, por ejemplo)
  - En el resto de casos usaremos **var**. Por ejemplo, en casos en los que las variables definidas tengan que salir de la función donde estén definidas.
- Por norma general, el uso de **var** queda reservado a casos muy especiales.

ES6 - Strings

- Los objetos de tipo String son uno de los tipos de datos más importantes en la programación.
- Se encuentran en la mayoría de lenguajes de programación de alto nivel y nos permiten llevar a cabo una serie de tareas de manera muy eficiente.
- En Javascript también representan un punto muy importante dentro del lenguaje y, los creadores de ES6 han decidido incrementar sus funcionalidades y agregar algunos casos de uso que faltaban.

ES6 - Strings

TEMPLATE LITERALS

- Se trata de la nueva forma que tenemos para poder incrustar expresiones dentro de una cadena de caracteres de una o varias líneas.
- La inclusión de la variable debemos hacerla usando los caracteres **${ }**.

```
const PERSONA = {
   nombre: 'Raul',
   edad: 22,
   direccion: 'C/ Falsa 123'
}

console.log(`Mi nombre es ${PERSONA.nombre},
tengo ${PERSONA.edad} años
y vivo en ${PERSONA.direccion}`)
```

ES6 - Strings

TEMPLATE LITERALS

- Para definir este tipo de plantillas usamos el carácter de **comillas invertidas**, en lugar de comillas simples o dobles.
- Podemos definir la ejecución de expresiones identificadas por el carácter **$** y limitadas por llaves. **${ expresión }**
- La ejecución de las expresiones contenidas entre llaves, junto con el texto definido alrededor de las mismas son enviados a una función que se encarga de concatenar el resultado de las primeras con el propio texto.
- Como hemos visto en el ejemplo anterior, podemos definir cadenas conteniendo múltiples líneas sin necesidad de usar el carácter especial **\n**.

ES6 - Strings

TEMPLATE LITERALS

- Dentro de estas plantillas podemos situar cualquier tipo de expresión:

```
function saludo(){
   return "Hola Mundo!"
}

console.log(`Plantilla con una operación: ${2 * 30}
y una llamada a una función: ${saludo()}`)
```

ES6 - Strings

TEMPLATE LITERALS

- Para trabajar de manera más avanzada con este tipo de plantillas, podemos definir una serie de funciones que se encarguen de postprocesar el literal.
- Con estas funciones seremos capaces de modificar la salida de las plantillas y devolver lo que nos interese en cada caso.
- Reciben como **primer argumento** un array con cada una de las cadenas de texto definidas dentro de la plantilla.
- El segundo y subsiguientes parámetros devuelven la ejecución de las diferentes expresiones incrustadas dentro del literal.
- Se puede establecer cualquier nombre a la función que apliquemos sobre la plantilla.

ES6 - Strings

TEMPLATE LITERALS

```
var x = 10
var y = 14

function mayus(strings, ...values){
   console.log(strings[0]) // 'Hola '
   console.log(strings[1]) // ' mundo '
   console.log(values[0])  // 140
   console.log(values[1])  // 24
   return strings[0].toUpperCase() + values[0] +
strings[1].toUpperCase() + values[1]
}

console.log(mayus`Hola ${x*y} mundo ${x+y}`)
// HOLA 140 MUNDO 24
```

ES6 - Strings

TEMPLATE LITERALS

- Disponemos además de la propiedad **raw** para poder trabajar con la cadena literal sin interpretar ningún carácter especial, como por ejemplo \n.

```
function nocar(strings, ...values){
   return strings.raw[0]
}

console.log(nocar`Hola\n mundo`)
// Hola\n mundo
```

ES6 - Clases

POO

- La **Programación Orientada a Objetos** es un paradigma de programación que utiliza la abstracción para crear modelos basados en el mundo real.
- Este paradigma utiliza diferentes técnicas entre las que se encuentra la modularidad, el polimorfismo o el encapsulamiento.
- Puede considerarse como el diseño de software en el que una serie de objetos cooperan entre sí para lograr un objetivo común.
- Cada objeto es capaz de recibir mensajes, procesar la información y enviar mensajes a otros objetos.
- Gracias a su gran modularidad, el código desarrollado mediante la POO ofrece una gran flexibilidad en el desarrollo y facilita el mantenimiento del software.

ES6 - Clases

POO

- Las **clases** son los esquemas que definen cómo vamos a crear nuestros objetos.
- Definen cuál serán las propiedades y el comportamiento de dichos objetos.
- Mediante la **instanciación** leemos estas propiedades y creamos tantos objetos como necesitemos para alcanzar el objetivo de nuestro software.
- Permiten abstraer los datos y sus operaciones asociadas a modo de **caja negra**.

ES6 - Clases

POO

- Un **objeto** es una unidad dentro de un programa que consta de un estado y un comportamiento.
- Los objetos se instancian siguiendo las instrucciones definidas en una clase e interactúan entre ellos enviándose mensajes.
- Definida una clase, podemos crear tantos objetos como sea necesario. Cada uno de ellos, aunque tendrá el mismo comportamiento, diferirá del resto en el valor de sus propiedades.

ES6 - Clases

- Hasta la llegada de ES6, para poder implementar clases y objetos en Javascript, podíamos proceder de la siguiente manera:

```
function Persona(nombre){
   this.nombre = nombre
}

var p = new Persona('Antonio')
console.log(p.nombre);
```

- Se define una función que recibe los parámetros de entrada para la creación de objetos y hace las veces de constructor.
- Los métodos asociados se agregan a través de **prototype**.

ES6 - Clases

- La sintaxis se ha modificado en ES6 para ser mucho más clara y aproximarse más a lo que encontramos en otros lenguajes.

```
class Persona{
    constructor(nombre){
        this.nombre = nombre
    }
}

let p2 = new Persona('Ramon')
console.log(p2.nombre);
```

- La definición es mucho más clara.
- Se utiliza el método **constructor** para poder inicializar las propiedades de los objetos a instanciar.

ES6 - Clases

- Podemos agregar los métodos de la clase incorporando las diferentes funciones dentro de la definición de la propia clase:

```
class Persona{
    constructor(nombre){
        this.nombre = nombre
    }

    hablar(){
        return 'Hola'
    }
}

var p = new Persona('Ramon')
console.log(p.hablar());
```

ES6 - Clases

- Mediante la palabra reservada **static** podemos definir los ‘métodos de clase’.
- Son aquellos métodos que están asociados directamente a la clase donde se definen y no es necesario disponer de una instancia para poder ejecutarlos.

```
class Persona{
    constructor(nombre){
        this.nombre = nombre
    }
    static tieneNombre(persona){
        if (persona.nombre !== undefined) return true
        return false
    }
}

var p = new Persona('Ramon')
console.log(Persona.tieneNombre(p));
```

ES6 - Clases

- Uno de los conceptos más importantes dentro de la POO es la **Herencia**.
- Una clase (hija) puede heredar todas las propiedades y métodos de otra clase diferente (padre).
- Aparte de la herencia de métodos y propiedades, puede definir los suyos propios, creando así una abstracción de la clase padre.
- En el siguiente ejemplo vemos cómo podemos crear una clase que extiende de la clase Persona planteada en ejemplos anteriores.

ES6 - Clases

```
class Adulto extends Persona{
    constructor(nombre, coche){
        super(nombre)
        this.coche = coche
    }
}

let a = new Adulto('María', true)
console.log(a.hablar()) // Hola
console.log(a.coche)    // true
Persona.tieneNombre(a)  // true
```

ES6 - Clases

- Mediante la palabra reservada **extends** especificamos cuál es la clase padre, de la que hereda todas sus propiedades y métodos.
- Con la llamada a **super** dentro del constructor de la clase hija, llamamos al constructor de la clase padre, donde ya se está inicializando la propiedad nombre.
- Dentro de la clase **Adulto** definimos propiedades específicas de esta clase, las cuales no estarán accesibles desde las instancias de la clase padre.
- Podríamos generar también métodos específicos únicamente para la clase Adulto.
- Podríamos decir que: *una instancia de la clase Adulto es también una Persona pero con sus propias características y acciones.*

ES6 - Clases

- Disponemos de los métodos **get** y **set** para asegurar la **encapsulación** de las propiedades de una clase.
- Cuando recuperamos el valor de alguna de las propiedades de nuestros objetos, es recomendable hacerlo a través de estos métodos y no dejar el acceso público directamente sobre los valores almacenados.
- Con esto nos aseguramos el control a la hora de recuperar los valores y asignarlos de nuevo.

ES6 - Clases

```
class Persona{
    constructor(nombre){
        this._nombre = nombre
    }

    get nombre(){
        return this._nombre
    }

    set nombre(nombre){
        this._nombre = nombre
    }
}

let p = new Persona('Rocio')
console.log(p.nombre) // Rocio
p.nombre = 'Raul'
console.log(p.nombre) // Raul
```

ES6 - Arrow Functions

- *'Arrow Functions'* es la nueva sintaxis que se ha incluído en ES6 para escribir nuestras funciones.
- Ahorran tiempo a los desarrolladores y simplifican la resolución del ámbito de las mismas.
- Sin duda, son una de las características nuevas más apreciadas por los desarrolladores Javascript.
- Este concepto consiste en definir nuestras funciones a través de los caracteres **=>**, simulando una flecha.
- Es el formato más idóneo para poder crear funciones anónimas que podamos almacenar en variables o pasar como parámetro de un método.
- Existe una gran variedad de sintaxis en función de lo que necesitemos construir, vamos a intentar ver las más interesantes.

ES6 - Arrow Functions

Sintaxis básica con múltiples parámetros

```
// ES5
let sumar = function(x, y){
    return x+y
}

// ES6
let sumar = (x, y) => {
    return x+y
}

console.log(sumar(3,5))
```

ES6 - Arrow Functions

Sintaxis básica con múltiples parámetros

```
let sumar = (x, y) => {
    return x+y
}
```

ES6 - Arrow Functions

Sintaxis básica con múltiples parámetros

```
let sumar = (x, y) => { return x+y }
```

ES6 - Arrow Functions

Sintaxis básica con múltiples parámetros

```
let sumar = (x, y) => x+y
```

ES6 - Arrow Functions

Sintaxis básica con un único parámetro

```
//ES5
let dividirStringES5 = function (cadena){
   return cadena.split(' ')
}

//ES6
let dividirStringES6 = cadena => cadena.split(' ')

console.log(dividirStringES5('Esto es una cadena'))
console.log(dividirStringES6('Otra cadena diferente'))
```

ES6 - Arrow Functions

Sintaxis básica sin parámetros

```
//ES5
let logErrorES6 = function(){
   console.log('Error en el programa')
}

//ES6
let logErrorES5 = () => console.log('Error en el
programa')

logErrorES5()
logErrorES6()
```

ES6 - Arrow Functions

- Este tipo de sintaxis es de gran utilidad en caso de tener que pasar por parámetro algún tipo de función.
- Es el caso de las funciones **map, reduce y filter**.

```
let numeros = [1,2,3,4,5]

numeros.map(num => console.log(num*2))

let pares = numeros.filter( num => num%2 == 0 )
console.log(pares)
```

ES6 - Manejo de Parámetros

- Se han ampliado una serie de características a la hora de interactuar con los parámetros de nuestras funciones.

Parámetros opcionales

- Se permite la asignación de un valor por defecto a los parámetros de un método, convirtiéndolos así en opcionales.

```
function operacion(x, y, z=3){
   return x*y*z
}

console.log(operacion(2,3))
console.log(operacion(2,3,5))
```

ES6 - Manejo de Parámetros

Rest Parameter

- Se pueden definir una serie de parámetros variables al final de la definición de los parámetros de una función.
- Para ello utilizamos el operador **spread** (...)
- Recibimos el parámetro en forma de **array**.

```
function unirCadenas(separador, ...palabras){
   return palabras.join(separador)
}

console.log(unirCadenas(',', 'uno', 'dos', 'tres'))
```

ES6 - Manejo de Parámetros

Arguments

- La palabra reservada **arguments**, utilizada dentro de una función, nos da acceso a los diferentes parámetros recibidos por dicha función.
- Podríamos recuperar los parámetros a través de arguments como si de un array se tratase, accediendo al índice que ocupan dentro de la definición de la función.

```
function f(param1, param2){
   console.log(arguments.length)
   console.log(arguments[0])
}

f('param1', 'param2')
```

ES6 - Manejo de Parámetros

Arguments vs Rest Parameters

- Los parámetros rest son aquellos a los que no se les ha asignado un nombre, mientras que el objeto arguments contiene todos los argumentos que recibe la función.
- El objeto arguments no es un array. Los parámetros rest son una instancia de la clase Array y por lo tanto, podemos aplicarles métodos como **sort, map o filter**.
- El uso de parámetros rest nos permiten reducir el código repetitivo que se usaba para acceder a los parámetros de nuestras funciones.

ES6 - Manejo de Parámetros

Destructuring

- Esta sintaxis específica nos permite extraer datos de arrays y objetos de una manera muy sencilla y ordenada.

```
let [a, b] = [1, 2]
console.log(a) // 1

let [c, d, ...rest] = [1,2,3,4,5,6]
console.log(rest) // [3,4,5,6]

let {e, f} = {e: 1, f: 2}
console.log(f) // 2

let {g, h, ...params} = {g: 4, h: 21, i: 8, j: 10, k: 31}
console.log(params) // {i: 8, j: 10, k: 31}
```

ES6 - Manejo de Parámetros

Destructuring

- Gracias a este concepto, las funciones pueden retornar múltiples valores y tratarlos de manera más sencilla que si lo hiciésemos a través de un array.

```
function f(){
   return [1,4]
}

let [a, b] = f()
console.log(`El valor de a: ${a} y el de b: ${b}`)
```

- En este caso, podríamos seguir recuperando la respuesta de la función dentro de un objeto de tipo Array

ES6 - Manejo de Parámetros

Destructuring

- En el ejemplo anterior, también podemos

```
function f(){
   return [1,4,7]
}

let [, b, c] = f()
console.log(`El valor de a: ${b} y el de b: ${c}`)
```

- Podríamos llegar incluso a ignorar todos los valores devueltos por la función.

ES6 - Manejo de Parámetros

Destructuring

- Puede ser interesante el uso de esta técnica para poder acceder de manera sencilla a objetos complejos de Javascript.
- Lo vemos más claro en el siguiente ejemplo:

ES6 - Manejo de Parámetros

Destructuring

```
function quienEs({nickname: nick, nombreCompleto: {nombre: nombre}}){
   console.log(`Su nombre es: ${nombre}, y su nick: ${nick}`)
}

var user = {
   id: 21,
   nickname: 'luna42',
   nombreCompleto: {
       nombre: 'Rosa',
       apellidos: 'Ramirez'
   }
}

quienEs(user)
```

ES6 - Módulos

- En la estructura básica de ES6, cada módulo se define en un fichero independiente.
- Las funciones o variables definidas dentro de cada uno de estos módulos no son visibles por otros módulos a no ser que exportemos su contenido.
- Para alcanzar este objetivo, disponemos de las palabras reservadas **import** y **export.**
- Vamos a analizar el comportamiento de esta funcionalidad a través de un ejemplo.

ES6 - Módulos

- Creamos el fichero **calculadora.js**

```
function suma(x, y){
   return x + y
}

function multiplicacion(x, y){
   return x * y
}

export {suma, multiplicacion}
```

- Mediante la palabra reservada **export** definimos qué partes de nuestro fichero van a poder ser importada por otros módulos.

ES6 - Módulos

- Dentro de nuestro fichero **app.js** podemos importar las funciones anteriores para poder ejecutarlas.

```
import {suma, multiplicacion} from './calculadora'

console.log(suma(4,3))
console.log(multiplicacion(3,8))
```

ES6 - Módulos

- Disponemos de diferentes maneras para poder importar las funciones de nuestros módulos.
- Una de las más comunes consiste en importar todas las funciones exportadas del módulo a través de un alias.

```
import * as Calc from './calculadora'


console.log(Calc.suma(4,3))
console.log(Calc.multiplicacion(4,5))
```

ES6 - Módulos

- Si dentro de un módulo en concreto queremos exportar un único valor, podemos utilizar la combinación **default export**.
- Modificamos el ejemplo anterior de la calculadora para ver cómo implementar este caso.

```
var operaciones = {
    suma: function(x, y){
        return x+y
    },
    multiplicacion: function(x,y){
        return x*y
    }
}

export default operaciones
```

ES6 - Módulos

- A la hora de importarlo, procederemos de la misma manera que anteriormente, pero teniendo en cuenta que todo queda encapsulado bajo el objeto que hemos exportado.

```
import operaciones from './calculadora'

console.log(operaciones.suma(4,3))
console.log(operaciones.multiplicacion(4,5))
```

- Podemos usar el mismo nombre del fichero de exportación o aplicarle uno nuevo.

ES6 - Promesas

- Las Promesas nos ofrecen una manera de procesar nuestras operaciones de manera asíncrona, sin perder el formato con el que estamos acostumbrados a trabajar.
- Ejecutan una función cuyo resultado nos será devuelto en algún momento del futuro.
- Además las promesas, al contrario que los callbacks, nos dan garantías de recibir dicho valor, aunque sea erróneo.

ES6 - Promesas

Creación de Promesas

- La forma más simple de crear nuestras propias promesas es a través de **new Promise**.
- Como parámetro del constructor pasamos la función que maneja cómo vamos a devolver el valor futuro.
- Esta función nos proporciona los métodos **resolve** y **reject**, las cuales se ejecutan para devolver el valor futuro y devolver un error respectivamente.
- Vemos un ejemplo simple:

ES6 - Promesas

Creación de Promesas

```
var p = new Promise(function(resolve, reject){
    if(/* condicion */){
        resolve(/* valor */)
    }else{
        reject(/* error */)
    }
})
```

ES6 - Promesas

Creación de Promesas

- Dentro de la creación de la promesa, lanzamos el método **resolve** con el valor que necesitemos devolver cuando la condición que establezcamos sea la correcta.
- Por el contrario, lanzaremos el método **reject** especificando el error devuelto cuando no se satisfaga la condición.
- Podemos incluso crear promesas que se resuelvan inmediatamente con **var p = Promise.resolve(32)**

ES6 - Promesas

Consumiendo de Promesas

- Una vez tenemos definida la promesa, disponemos de una serie de métodos que nos van a permitir recuperar los valores devueltos o los errores en caso de haberlos.
- El primer método, ejecutado sobre la instancia de la promesa es **then**.
- Este método recibe una función como parámetro donde tendremos la resolución de la promesa con la que estamos trabajando.

```
var p = new Promise((resolve, reject) => resolve(58))
p.then( (val) => console.log(val) ) // 58
```

ES6 - Promesas

Consumiendo de Promesas

- El método **then** puede recibir dos parámetros.
- El segundo, nos devuelve el error que pueda suceder de la ejecución de la promesa, es decir, la llamada al método **rejected.**

```
var p = new Promise((resolve, reject) => resolve(58))
p.then( (val) => console.log(val),
        (err) => console.log(err))
```

ES6 - Promesas

Consumiendo de Promesas

- Si queremos ser más precisos en el manejo de errores dentro de la ejecución de una promesa, debemos utilizar el método **catch**.
- En la concatenación de varios métodos sobre una promesa, podemos usar una única llamada al método **catch** para gestionar cualquier tipo de error que aparezca.
- El ejemplo anterior, gestionado con catch, sería el siguiente:

```
var p = new Promise((resolve, reject) => resolve(58))
p.then( (val) => console.log(val))
.catch( (err) => console.log(err))
```

ES6 - Promesas

Consumiendo de Promesas

- Podemos ver en este ejemplo cómo se comporta el método catch, forzando la excepción.

```
var p = new Promise((resolve, reject) => {
   if(true){
       throw new Error("Excepción dentro de la promesa")
   }else{
       resolve(12)
   }
})

p.then( val => console.log(val))
.catch( err => console.log("Error: " + err.message))
```

ES6 - Promesas

Consumiendo de Promesas

- El método catch podría recuperar cualquier excepción sucedida en las sucesivas llamadas a then a partir de la instancia de una promesa.

```
var p = new Promise((resolve, reject) => {
   resolve(59)
})

p.then((val) => console.log(val))
   .then( val => {throw new Error("Falla en el paso 2")})
   .then( val => console.log(val))
   .catch( err => console.log("Error: " + err.message))
```

ES6 - Promesas

Concatenando Promesas

- Existen ocasiones en las que necesitamos controlar la ejecución de varias promesas.
- Para ello tenemos el método **all**, mediante el cual gestionamos con una única llamada al método **then** el resultado de todas las promesas.
- De igual manera, con una llamada al método **catch** podemos recoger todos los errores que vayan apareciendo.
- Lo vemos a partir de un ejemplo:

ES6 - Promesas

Concatenando Promesas

```
var promises = []

var p1 = Promise.resolve(34)
var p2 = Promise.resolve(21)
promises.push(p1)
promises.push(p2)
Promise.all(promises)
    .then( results => {
        results.forEach( item => {
            console.log(item)
        })
    })
    .catch( err => {
        console.log("Err: "+err)
    })
```