

# UD9. AVANZADOS

# CONCEPTOS

|   |            |
|---|------------|
| 1.JAVASCRIPT  | AVANZADO   |
| .....   | 2          |
| 2.CONTROL   | DE ERRORES |
| .....   | 2          |
| 2.1. INTRODUCCIÓN AL CONTROL DE ERRORES .....   | 2          |
| 2.2. ERRORES, EXCEPCIONES Y AVISOS .....  | 3          |
| 2.3. JAVASCRIPT MÁS Estricto.....   | 3          |
| 2.4. CREAR Y LANZAR ERRORES PROPIOS .....   | 4          |
| 2.5. GESTIONAR LAS EXCEPCIONES. BLOQUE TRY ...CATCH .....   | 4          |
| 3. ....   | MÓDULOS    |
| .....   | 6          |
| 3.1. USO DE MÓDULOS Y PAQUETES.....   | 6          |
| 3.2. CARGA Y CREACIÓN DE MÓDULOS .....  | 7          |
| 3.2.1. CREACIÓN DE MÓDULOS .....  | 7          |
| 3.2.2. CARGA DE MÓDULOS .....   | 8          |
| 4.PROGRAMACIÓN  | ASÍNCRONA  |
| .....   | 9          |
| 4.1. PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA.....   | 9          |
| 4.2. CALLBACK HELL.....   | 10         |
| 4.3. PROMESAS.....  | 10         |
| 4.3.1. INTRODUCCIÓN A LAS PROMESAS.....   | 10         |
| 4.3.2. CREACIÓN DE PROMESAS .....   | 11         |
| 4.3.3. MÉTODO THEN .....  | 12         |
| 4.3.4. MÉTODO CATCH .....   | 12         |
| 4.3.5. EJEMPLOS DE PROMESAS .....   | 12         |
| 4.3.6. ENCADENAMIENTO DE MÉTODOS .....  | 14         |
| Se puede invocar varias veces a las funciones de resolución y de rechazo. Pero cada método tehn o catch solo puede responder a una. Lo que sí se permite es encadenar los métodos then y catch las veces que haga falta. Eso es una buena manera de estructurar nuestras aplicaciones evitando un exceso de control de funciones callback. .... |            |
| 4.3.7. MÉTODOS DEL OBJETO PROMISE .....   | 15         |

## 1.JAVASCRIPT AVANZADO

El lenguaje JavaScript ha mejorado enormemente para acomodarse a las necesidades planteadas por los desarrolladores de aplicaciones web. Esta unidad presenta algunos de estos elementos más celebrados en las mejoras del lenguaje.

Todo desarrollador debe tener en cuenta que el lenguaje JavaScript es un lenguaje vivo que se sigue ampliando y mejorando continuamente. Las novedades tardan un tiempo en ser adoptadas al 100% por los navegadores, las que vamos a ver han sido adoptadas por todos ellos, salvo los que, como IE, no siguen actualizándose.

## 2.CONTROL DE ERROES

### 2.1. INTRODUCCIÓN AL CONTROL DE ERRORES

Los errores que ocurren en un programa pueden ser:

- Errores al escribir código por parte del programador. Son **errores de sintaxis**, que son los más fáciles de detectar porque cuando se interpreta el código, se nos indica el error. En el caso de que el código se esté creando en un entorno de trabajo avanzado, hay errores que aparecen marcados en el mismo instante en el que hemos escrito el código. En el caso de las aplicaciones web, el error aparece también marcado en la consola del panel de depuración del navegador.

En realidad hay 2 tipos:

- ✓ Errores detectables en tiempo de escritura. Fallos de sintaxis evidentes que en la mayoría de entornos de desarrollo (incluido Visual Studio Code) pueden marcar antes de probar el código. Muchas veces estos entornos subrayan el código erróneo en rojo.
- ✓ Errores de ejecución. Son errores que solo se pueden detectar cuando el código se intenta ejecutar para probar la aplicación. Un ejemplo de este tipo es cuando en el código se invoca a una función que aún no ha sido definida. Solamente cuando tratamos de ejecutar el código se puede detectar que esa función no existe.
- Hay errores por mala lógica al desarrollar la aplicación. Se **errores lógicos**, en los que la sintaxis es correcta, pero el programa no funciona como debería. Ninguna herramienta nos avisa automáticamente del error, aunque sí hay herramientas especiales que facilitan su detección, es el desarrollador el que detecta que la aplicación no funciona como debería, o bien, los propios usuarios.
- Hay errores por causas externas. Son **errores del sistema**, circunstancias que provocan el error pero que están fuera del control del programador: fallo en la conexión de red, caída de un servicio que estábamos utilizando, etc. No podemos controlar estos fallos la mayoría de veces, pero al menos sí podemos matizar el daño que causan a nuestra aplicación.

- **Errores de usuario.** Son los provocados por las acciones inesperadas que realiza el usuario y que causan un error en tiempo de ejecución. Por ejemplo, pedir al usuario un número y recibir texto. En realidad, son errores lógicos que ocurren por no prever estas situaciones.

## 2.2. ERRORES, EXCEPCIONES Y AVISOS

Un **error** es un fallo que produce el programa y que tiene como consecuencia que la aplicación se detenga. Los errores no están controlados y provocan todo tipo de situaciones indeseadas en la ejecución de la aplicación.

Una **excepción** es un error que podemos controlar para que se gestione adecuadamente. Las excepciones permiten manejar objetos especiales que contienen los detalles del error para poder lidiar con el mismo de la mejor manera posible.

Un **aviso (warning)** es un error que se considera leve. No impide la ejecución del programa, pero, al menos, intenta avisar del problema al desarrollador para que conozca la situación. Los avisos pueden ser vitales para detectar, de forma temprana, errores complejos de resolver

## 2.3. JAVASCRIPT MÁS ESTRICTO

JavaScript se ideó bajo una capa de rapidez y dinamismo que hizo que este lenguaje no tuviera una sintaxis rígida. Las ventajas de este hecho son: facilidad para empezar a desarrollar aplicaciones con este lenguaje y el dinamismo que se consigue en los resultados con poco esfuerzo de escritura de código.

El problema es que, a medida que las posibilidades de las aplicaciones JavaScript han aumentado, el control de errores se ha hecho cada vez más importante y el lenguaje original, en este sentido, no ayudaba mucho.

Un ejemplo de variante más estricta es el lenguaje creado por Microsoft con el nombre de **TypeScript**. Este lenguaje tiene una sintaxis más formal y estricta que es más del gusto de muchos desarrolladores, es famoso porque añade muchos tipos de datos al lenguaje y fuerza que se utilicen los elementos del lenguaje de forma más estricta, especialmente en todo lo referente al uso de tipos de datos.

**TypeScript** no tuvo un gran éxito hasta que se convirtió en el lenguaje base del framework **Angular** de **Google** que es ampliamente utilizado por una enorme comunidad de desarrolladores. **TypeScript** es más potente para detectar errores, pero es más pesado de escribir al ser mucho más rígido. Ningún navegador entiende **TypeScript**, por lo que su código debe convertirse a JavaScript con la ayuda de un software especial.

No obstante, desde la versión ES5 del estándar, se puede activar en JavaScript el llamado **modo estricto**, que es un modo más exigente con los errores. Por ejemplo, si ejecutamos este código por consola:

```
'use strict';  
x=9;  
console.log(x);
```

Ahora sí se produce un error, indicando que x no está definida. El modo estricto se puede activar para todo un archivo, como en el ejemplo anterior, pero se puede activar para el código interior a funciones concretas:

```
function f(param){
    'use strict';
    ...
}
```

## 2.4. CREAR Y LANZAR ERRORES PROPIOS

Los errores se crean por parte del intérprete de JavaScript cuando ocurren. Pero podemos crear nuestros propios errores:

```
let miError=new Error("Se esperaba un número");
```

La variable `miError` es una referencia a un objeto que representa un error. El objeto `Error` representa errores genéricos, pero hay 6 objetos que sirven para crear errores propios de tipo más específico.

| TIPO DE ERROR               | USO   |
|-----------------------------|---|
| <code>EvalError</code>      | Error al intentar usar la función <code>eval()</code> de JavaScript |
| <code>RangeError</code>     | Error numérico o de rango de valores incorrecto                     |
| <code>ReferenceError</code> | Referencia a objeto no válido                                       |
| <code>TypeError</code>      | Error por mal uso de los tipos de datos                             |
| <code>URIError</code>       | Error al codificar o decodificar la URL                             |

En base a la tabla anterior sería más correcto:

```
let miError=new RangeError("Se esperaba un número");
```

Crear un objeto de error no provoca ningún error. Lanzar el error en sí y provocar una excepción, se hace con la palabra clave **throw**:

```
let miError=new RangeError("Se esperaba un número");
throw miError;
```

Se provoca un error que podrá ser analizado por la consola.

## 2.5. GESTIONAR LAS EXCEPCIONES. BLOQUE TRY ...CATCH

JavaScript aporta una estructura llamada **try..catch**, que trabaja con la siguiente sintaxis:

```
try{
    ...
    código que puede provocar un error
    ...
}catch{
    ...
    código que se ejecuta si hay error
    ...
}
```

En el bloque encabezado por la palabra **try** se coloca el código que puede provocar un error. Si ese error se produce, el flujo del programa pasa al apartado **catch**, dejando el resto de líneas del **try** posteriores a la que produce el error, sin ejecutar.

Veamos con un ejemplo:

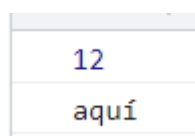
```
try{
    console.log(e);
    console.log("aquí");
}catch(error){
    let e=1;
    console.log(e);
}
```

Si el archivo solo tiene este código, se va a intentar escribir por consola el valor de una variable llamada **e** que no existe porque no se ha declarado. Con lo cual el código **console.log(e)** produciría un error. Al estar en un bloque **try**, se crea el objeto de error y se envía dicho objeto al apartado **catch**. Las líneas dentro del **catch** pasan a ejecutarse: se declara la variable **e** con valor 1 y se escribe por pantalla. Veremos al ejecutar el código, que se muestra el número 1 por consola.

Todo cambia si **e** se declara antes de la instrucción **try**:

```
let e=12;
try{
    console.log(e);
    console.log("aquí");
}catch(error){
    let e=1;
    console.log(e);
}
```

Se muestra por consola:



```
12
aquí
```

No se muestra el código del **catch**, no hay error.

Hay posibilidad de añadir un bloque **finally** cuyo contenido se muestra tanto si se produce una excepción, como si no:

```
try{
    console.log(e);
    console.log("aquí");
}catch(error){
    let e=1;
    console.log(e);
}
finally{
```

```
    console.log("Yo salgo siempre");  
  }  
  1  
  Yo salgo siempre  
  ,
```

## 3. MÓDULOS

### 3.1. USO DE MÓDULOS Y PAQUETES

La programación de aplicaciones para **node.js** ha conseguido influir mucho las nuevas normas sobre JavaScript. Una de sus más llamativas aportaciones al JavaScript clásico ha sido el uso de módulos. **Node.js** incorpora un gestor de paquetes llamado **npm**. En la programación front-end el uso de **npm** está orientado principalmente a la instalación de utilidades interesantes que nos ayudan en el proceso de desarrollo.

Cada paquete integra una serie de módulos, entendiendo un módulo como una serie de funciones, constantes, objetos, etc que se usan como una librería que podemos reutilizar para el desarrollo de aplicaciones. Los módulos permiten de forma eficaz tanto utilizar módulos de librerías de terceros, como crear módulos propios que pueden reutilizarse en todas nuestras aplicaciones, permitiendo condensarlas más y facilitar la reutilización de código.

Las aplicaciones de node.js hacen uso de módulos desde hace mucho tiempo, pero para el JavaScript del lado cliente ¿se pueden utilizar los módulos?

Lo cierto es que del lado cliente, no se usaban módulos, estaban fuera de la norma. Si deseamos usar librerías de funciones y otros elementos, estas se cargan mediante etiquetas de script:

```
<script src= "libreria1.js "></sript>  
<script src= "libreria2.js "></sript>  
<script src= "libreria3.js "></sript>  
...  
<script>  
... código que usa funciones de las librerías anteriores...  
</script>
```

El código anterior muestra la forma de trabajar cuando queremos reutilizar código JavaScript o utilizar librerías de terceros (como jQuery, por ejemplo). La instrucción script, cuando cargar archivos externos, genera una petición http por cada librería y eso puede ralentizar la ejecución de la aplicación.

Con los años, muchos desarrolladores empiezan a echar mucho de menos un sistema de módulos como el de node.js para el desarrollo de aplicaciones en el lado cliente.

Teniendo en cuenta lo anterior, han aparecido numerosas utilidades para añadir algún mecanismo de carga de módulos para el JavaScript del navegador. Por ejemplo: **Require.js**<sup>1</sup> y **System.js**<sup>2</sup> que requieren de la cara de un archivo JavaScript con las funciones que aportan para poder importar módulos en nuestro código. Es decir, son librerías que nos proporcionan una forma (no estándar) de importar módulos.

Otras opciones pasan por utilizar precompiladores que permiten que en nuestro código usemos instrucciones de carga de módulos y a través de herramientas de compilación, ese código se traduce a un código final que dispondrá de módulos cargados. Ejemplos de esta técnica son las utilidades **Browserify** o **webpack**.

## 3.2. CARGA Y CREACIÓN DE MÓDULOS

La norma **ES2015 (ES6)** al fin incorporó el uso de módulos de JavaScript. Con esta norma aparecieron las palabras claves **export** e **import**.

### 3.2.1. CREACIÓN DE MÓDULOS

Cuando deseemos en nuestro código alguno de los elementos de un módulo bastará con indicar qué módulo queremos importar y qué elementos concretos deseamos de él (o bien importar todo).

Por otro lado, cuando se crea un módulo también hay que indicar qué cosas son importables. Supongamos que estamos creando un módulo llamado **geometria.js** en el que deseamos colocar funciones de cálculo de áreas y perímetros de figuras. En ese archivo hemos creado la función **areaCirculo** en base a lo siguiente:

```
export function areaCirculo(radio){  
    return Math.PI * radio * radio;  
}
```

La palabra **export** hace referencia a que esa función es exportable en otro archivo JavaScript.

Cada función o variable que queremos exportar debe tener por delante la palabra **export**.

```
export const NUMEROPI=Math.PI;  
export function areaCirculo(radio){  
    return NUMEROPI * radio * radio;  
}  
export function areaCuadrado(lado){  
    return lado**2;  
}
```

También podemos acumular todo lo que queremos exportar en una sola instrucción **export**:

```
const NUMEROPI=Math.PI;  
function areaCirculo(radio){  
    return NUMEROPI * radio * radio;  
}
```

---

<sup>1</sup> Disponible en <https://requirejs.org/>

<sup>2</sup> Véase <https://github.com/systemjs/systemjs>

```
function areaCuadrado(lado){  
    return lado**2;  
}  
export{  
    NUMEROPI;  
    areaCirculo;  
    areaCuadrado;  
}
```

Aquellos elementos del módulo que no estén en la instrucción **export** se considerarán privados y, por lo tanto, no exportables

### 3.2.2. CARGA DE MÓDULOS

La importación de módulos se realiza con la instrucción **import**. Esta instrucción debe de ser la primera (puede haber varias instrucciones **import**) del código JavaScript. Si queremos cargar un elemento del módulo, podremos hacer lo siguiente:

```
Import { areaCirculo } from "./geometria.js ;  
console.log(areaCirculo(5));
```

El código anterior carga la función **areaCirculo** del módulo que se ha configurado en el archivo **geometria.js**. Si queremos importar varios elementos del módulo, estos se separan con comas:

```
Import { areaCirculo, NUMEROPI } from "./geometria.js ;  
Podemos renombrar los elementos importados:
```

```
Import { areaCirculo as Circulo, areaCuadrado as Cuadrado } from "./geometria.js ;  
console.log(areaCirculo(9)); //Escribe 88.82643960980423  
Console.log(areaCuadrado(4)); //Escribe 16
```

También podemos importar todos los elementos del módulo, pero tenemos que asignar un nombre que se utilizará como espacio de nombre. Los espacios de nombre son un identificador que se usa como prefijo delante del nombre de cada elemento importado (función, método, variable, etc) La razón de su uso es diferenciar los nombres de elementos pertenecientes a 2 módulos distintos.

```
Import * as geom from "./geometria.js ;  
console.log(geom.areaCirculo(9));  
console.log(geom.areaCuadrado(4));
```

Un detalle importante, es que en el código HTML, si nuestro código JavaScript utiliza módulos, la etiqueta **script** que contiene ese código o que carga el código desde un archivo debe utilizar el atributo **type** con el valor **module**. Es decir, el código anterior completo, sería:

```
<script type="module">  
Import * as geom from "./geometria.js ;  
console.log(geom.areaCirculo(9));  
console.log(geom.areaCuadrado(4));  
</script>
```



## 4.PROGRAMACIÓN ASÍNCRONA

### 4.1. PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA

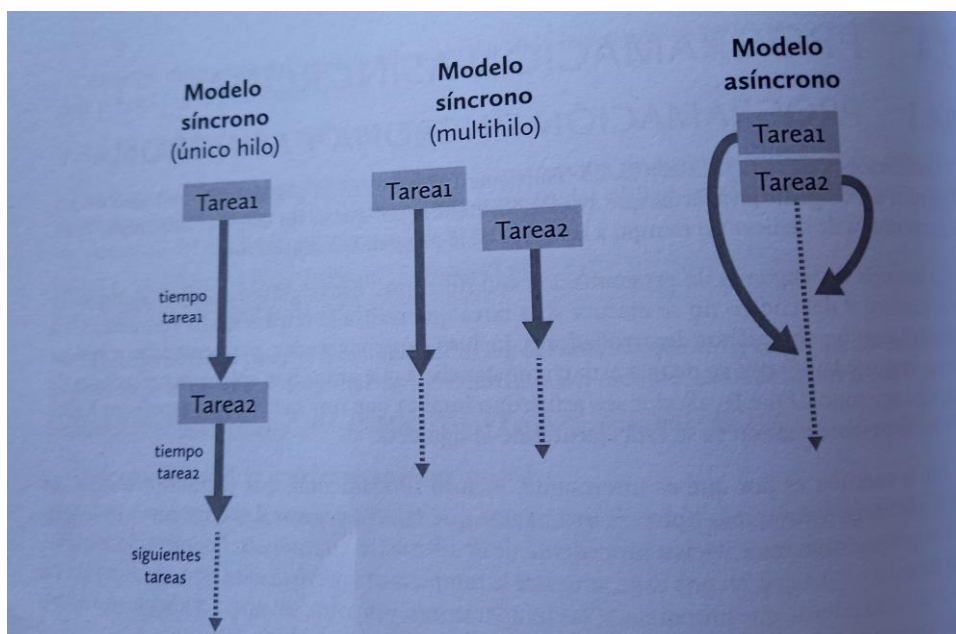
La mayoría de los lenguajes de programación son síncronos. Esto significa que, por ejemplo, la línea número 9 del código no se ejecuta si la tarea que realiza la línea número 8 no ha terminado. La dificultad que tienen muchos desarrolladores, incluso experimentados, para entender y aplicar bien el lenguaje JavaScript se debe a esta circunstancia: a que antes han sido programadores de lenguajes síncronos.

Una página puede necesitar cargar 2 elementos independientes que muestran información procedente de 2 servicios de internet. Por ejemplo, podríamos crear una página que, en una capa, muestre la temperatura prevista para hoy, proporcionada por un servicio externo que proporciona las temperaturas, y en otra, un mapa que muestre cómo llegar a nuestra oficina, usando un servicio de mapas de Internet.

La temperatura y el mapa tardan en cargar. Si la programación fuera síncrona, el mapa no se carga si antes no ha llegado la información de la temperatura. No nos interesa este efecto, es más eficiente que ambos componentes se carguen de forma independiente.

En lenguajes síncronos la única manera de conseguir este tipo de efectos es crear varios hilos independientes en los que cada uno realiza una tarea. Así funciona, por ejemplo, el lenguaje Java para conseguir 2 procesos que consigan resultados de forma independiente. JavaScript es un lenguaje de un único hilo, pero las operaciones sobre la red y otras operaciones de E/S (como consultar bases de datos) se lanzan de forma independiente.

Pero también este efecto tiene sus problemas. ¿Y si deseamos colorear el mapa?. Debemos asegurarnos de que el mapa ha llegado antes de empezar a colorear. Esto requiere sincronización, algo del tipo: cuando el mapa llegue, coloreamos.



Solución a esto ya tenemos, puesto que conocemos las funciones callback. Mediante funciones callback podemos hacer que el código de colorear se lance justo cuando se ha cargado el mapa. Algo, como esto:

```
componente.haCargado(())=>{colorear()});
```

Se invoca a colorear cuando el componente se ha cargado.

## 4.2. CALLBACK HELL

Este término (infierno de los callback) se popularizó hace unos años cuando se percibió el problema de tener que sincronizar numerosas acciones. Esto provoca un exceso de funciones **callback** que van desplazando el código a su derecha haciéndole poco legible y mantenible.

Siguiendo con el ejemplo anterior del componente de mapas, supongamos que ahora deseamos que el mapa se cargue cuando hagamos clic a un botón concreto. Además, tras colorear el mapa, deseamos realizar una animación sobre el mismo. Pero, todo ello, solo si hemos podido realizar correctamente y sin error las tareas anteriores. El código podría ser:

```
botón.addEventListener("click",function(ev){
  cargarMapa(componente, function(error){
    if(error){
      console.log("Error al cargar el mapa");
    }else{
      colorear(componente, function(error){
        if(error){
          console.log("Error al colorear");
        }
      })
    }else{
      animar(Componente);
    }
  })
})
});
```

Con tantas llamadas de tipo callback, el código se desplaza mucho a la derecha. En este código hemos hecho que las funciones callback reciban una variable que marca si hubo error en la operación. El hecho de valorar errores es fundamental para saber si el proceso se hizo bien o mal, es muy importante en labores de programación asíncrona. Pero esa valoración, que es habitual, desplaza aún más el código.

No hay duda de que la invocación `animar(componente)` realizará la animación tras colorear. Por otro lado, colorear se realiza tras la carga del mapa, y la carga no se inicia si no se ha hecho clic en el botón. El planteamiento es correcto, pero la legibilidad es terrible (Un infierno)

## 4.3. PROMESAS

### 4.3.1. INTRODUCCIÓN A LAS PROMESAS

La solución al problema anterior es una estructura que permite controlar de forma más organizada las tareas asíncronas sin tener que usar tantas funciones callback anidadas. Esa nueva estructura es lo que se conoce como promesa.

Las promesas, como tantas veces ocurre con JavaScript, se implementaron primero en librerías de terceros. Las más famosas son **Q**<sup>3</sup>, **When**<sup>4</sup>, el objeto Promise de la librería **WinJS** de Microsoft y la que ha resultado más influyente **RVSP.js**<sup>5</sup>. Se siguen usando por parte de algunos desarrolladores y algunas son respetuosas con el estándar, como RVSP.js, aportando métodos extras que algunos desarrolladores agradecen mucho.

La norma **ES2015** finalmente incorporó las promesas en el estándar y, desde entonces, se utilizan extensamente por parte de la comunidad de desarrolladores.

Una promesa permite invocar a una función o tarea, cuya labor requiere una ejecución asíncrona. Podremos determinar lo que ocurre en el caso de que esa tarea concluya correctamente y lo que haremos en el caso de concluir mal. En las 2 situaciones, podremos recoger información al respecto.

Las promesas pueden tener uno de estos estados:

- Cumplida (**resolved** o **fulfilled**). Si la tarea relacionada con la promesa se ha finalizado con éxito.
- Rechazada (**rejected**). Si la tarea no finaliza con éxito
- Pendiente de finalizar (**pending**). En proceso de finalización
- Finalizada (**settled**). Independientemente de si se ha cumplido o no, la tarea relacionada con la promesa ha finalizado.

Las promesas son objetos de tipo **Promise**. Estos objetos son los que hacen la labor de relacionar la labor asíncrona con las acciones a tomar en caso de éxito o fracaso. Para ello, proporcionan varios métodos y, sobre todo, una función de construcción de objetos **Promise** que es donde se constituye realmente la promesa.

### 4.3.2. CREACIÓN DE PROMESAS

La creación de promesas sigue esta estructura formal:

```
let promesa = new Promise(function(resolver, rechazar){
    código de la tarea asíncrona
    if(condición que valida que la tarea fue exitosa){
        resolver(información sobre el éxito);
    }else{
        rechazar(información sobre el fracaso);
    }
});
```

Para explicar el código anterior veamos lo siguiente:

- Las promesas son objetos que se construyen indicando una función de tipo callback.
- Esa función callback acepta 2 parámetros, los cuales son 2 funciones. La primera (se suele llamar **resolve**) se invoca cuando se ha verificado que la operación ha finalizado de forma correcta. La

<sup>3</sup> <https://github.com/kriskowal/q>

<sup>4</sup> <https://github.com/cujojs/when>

<sup>5</sup> <https://github.com/tildeio/rsvp.js>

segunda (se suele llamar **reject**) se invoca cuando se ha determinado que el proceso no ha finalizado correctamente.

- Ambas funciones reciben parámetros. A cada una se le envía un objeto que contiene la información de la resolución en caso de éxito (función **resolve**) o de fracaso (**reject**). Para la función de rechazo el parámetro suele ser un objeto de error, ya que permite la gestión de errores de forma más conveniente.

La creación del objeto promesa implica lanzar la tarea en segundo plano.

### 4.3.3. MÉTODO THEN

Pero la cuestión no es cuándo recogemos los resultados del éxito o el fracaso. Ahí interviene el método **then**. Este método acepta una función callback, que será invocada cuando la tarea de la promesa finalice con éxito. Hay un segundo parámetro opcional, que también es una función, cuyo código se invoca si el resultado es erróneo. La sintaxis es la siguiente:

```
promesa.then(function(resultado){...}, function(error){...});
```

La primera función recibe un parámetro. Ese parámetro es el que hayamos pasado a la función **resolver**, vista en el apartado anterior, durante la creación de la promesa. La segunda función recibe el parámetro indicado en la función **rechazar**. Es poco habitual usar la segunda función, en su lugar se usa el formato de captura de errores que veremos a continuación.

### 4.3.4. MÉTODO CATCH

En el caso de la función **rechazar**, es habitual lanzar un error. Por eso, hay un método llamado **catch** que permite gestionar el rechazo en la promesa. Es un formato mucho más coherente y que se asemeja a la estructura **try..catch**. Además, se permite encadenar ambos métodos, porque el resultado de los métodos **then** y **catch** es el propio objeto de la promesa (objeto **Promise**). La sintaxis completa es:

```
promesa.then(function(resultado){  
  ...  
}).catch(function(resultado){  
  ...  
});
```

En ese código podemos encadenar ambas acciones. Incluso hay un tercer método llamado **finally** que recibe una función callback sin parámetros, cuyo código se ejecuta tanto si la promesa es exitosa como si no.

### 4.3.5. EJEMPLOS DE PROMESAS

La idea es simple, pero es compleja de entender dada su novedosa sintaxis. Por ello, vamos a empezar haciendo una promesa simple. Haremos una promesa que consiste en que, si dos variables valen lo mismo, por pantalla salga un mensaje diciéndolo, y si no, lanzaremos un error. Evidentemente para una acción como esta, no hace falta usar promesas, pero usamos esta idea para entender el funcionamiento de las promesas.

```
var promesa=new Promise((resolver, rechazar)=>{  
  let n1=2;  
  let n2=2;
```

```
if(n1==n2) resolver("¡Son iguales!");  
else rechazar(Error("Algo raro ha pasado"));  
});  
promesa.then((respuesta)=>{  
  console.log(respuesta);  
});
```

Instantáneamente aparece el texto “¡Son iguales!” porque la condición es verdadera y eso provoca lanzar la función **resolver** enviando como parámetro el texto “¡Son iguales!”. El método **then** lanza la función callback que usa como parámetro el texto anterior y ese texto sale por pantalla, ya que la función callback del método **then** solo hace la labor de mostrar el parámetro que reciba por consola.

Si cambiamos los valores de **n1** y **n2** para que no sean iguales, se provocará una excepción que paraliza el programa y muestra el error. Como no hemos capturado errores, la ejecución de la aplicación finaliza.

Para capturar el error y gestionarlo, es para lo que se usa **catch**. La estructura completa sería esta:

```
var promesa=new Promise((resolver, rechazar)=>{  
  let n1=3;  
  let n2=2;  
  if(n1==n2) resolver("¡Son iguales!");  
  else rechazar(Error("Algo raro ha pasado"));  
});  
promesa.then((respuesta)=>{  
  console.log(respuesta);  
}).catch((error)=>{  
  console.log(error.message);  
});
```

Los valores de **n1** y **n2** son distintos lo que provoca que invoquemos la función para el rechazo. A esa función le pasamos como parámetro un objeto de error. El método **catch** ejecuta su función callback a la que se le pasa un objeto de error. El método **catch** ejecuta su función callback a la que se le pasa el objeto de error (sin que la excepción se lleve a cabo) y entonces, simplemente (gracias al método **message** de los objetos de error) se mostrará por consola el mensaje “Algo raro ha pasado”.

Veamos este otro ejemplo:

```
var promesa=new Promise((resolver, rechazar)=>{  
  throw new Error("La he liado parda");  
});  
  
promesa.then((respuesta)=>{  
  console.log("Me ha ido bien");  
}).catch((error)=>{  
  console.log("Me ha ido mal");  
});
```

En la creación de la promesa, a las primeras de cambio, lanzamos un error. Aunque no se ha gestionado nada con las funciones **resolver** y **rechazar**, lo cierto es que se ha generado un error, y ese error provocará que se ejecute la función callback del método **catch**. Es decir, por pantalla aparece el texto **Me ha ido mal**.

Veamos un ejemplo que usa temporizadores:

```
var promesa=new Promise(function(resolver, rechazar){
let n=0;
let intervalo=setInterval(function(){
n++;
    if(n==10){
        resolver("Han pasado 10 segundos");
clearInterval(intervalo);
    }
},1000);
});
```

```
promesa.then(function(mensaje){
console.log(mensaje);
});
```

En este código, en la creación de la promesa se genera un intervalo cuya función interna es invocada cada segundo. Un contador (n) va incrementándose en cada llamada. Cuando este contador llega a 10, invoca a la función **resolver** con el mensaje **Han pasado 10 segundos**. El método **then** será invocado entonces y mostrará ese mensaje por consola.

#### 4.3.6. ENCADENAMIENTO DE MÉTODOS

Se puede invocar varias veces a las funciones de resolución y de rechazo. Pero cada método **then** o **catch** solo puede responder a una. Lo que sí se permite es encadenar los métodos **then** y **catch** las veces que haga falta. Eso es una buena manera de estructurar nuestras aplicaciones evitando un exceso de control de funciones callback.

Un ejemplo sencillo de esta idea sería este:

```
var promesa=new Promise(function(resolver, rechazar){
let n=0;
let intervalo=setInterval(function(){
n++;
    if(n==10){
        resolver("Han pasado 10 segundos");
clearInterval(intervalo);
    }
},1000);
});
```

```
promesa.then(function(mensaje){
console.log(mensaje);
    return "Se ha cerrado el temporizador";
}).then((mensaje)=>console.log(mensaje));
```

A los 10 segundos se muestra Hemos llegado a 10 y justo debajo el texto Se ha cerrado el temporizador. El hecho de que el primer **then** use un **return**, provoca que se devuelva una nueva promesa, cuya función **then**, en este caso, se resuelve al instante, mostrando el resultado que hemos comentado.

Con estos códigos parece que estemos simplemente complicando el lenguaje, pero sin utilidad alguna. Será en la siguiente unidad cuando veamos la utilidad práctica real de las promesas y el encadenamiento de las mismas.

Si volvemos al ejemplo de colorear un mapa, no podemos colorear el mapa si no lo hemos podido cargar. Supongamos, además, que debemos cargar para colorear, una plantilla de colores de Internet. Si esa plantilla no podemos colorear. Podríamos crear una promesa asociada al éxito de cargar o no el mapa. Si se carga de forma correcta el mapa, entonces lanzamos el segundo método de cuyo éxito depende el coloreado final.

La idea con promesas sería la siguiente:

```
cargarMapa()  
.then((mapa)=>cargarPlantilla(mapa))  
.then((mapa)=>colorear(mapa));  
.catch(throw new Error("Error en la carga"));
```

Este código queda limpio, pero la potencia la dan las funciones **cargarMapa**, **cargarPlantilla** y **colorear**. El resultado de las 2 primeras tiene que ser una promesa que indique la correcta finalización de la tarea o su fallo al finalizar.

#### 4.3.7. MÉTODOS DEL OBJETO PROMISE

El objeto **Promise** aporta un método estático llamado **Promise.resolve**, que crea una nueva promesa resuelta y enviando (como si se hubiera invocado al método **resolver** en la creación de la promesa) el objeto que se envíe como parámetro.

Ejemplo de uso:

```
let promesa=Promise.resolve("Ha funcionado todo");  
promesa.then((mensaje)=>{console.log(mensaje)});
```

Se muestra: **Ha funcionado todo**. Luego, lo que hace este método es crear una promesa cumplida.

El método contrario es **Promise.reject**:

```
let promesa=Promise.reject(Error("No ha funcionado nada"));  
promesa  
  .then((mensaje)=>{console.log(mensaje)});  
  .catch((error)=>{console.log(error.message)});
```

Muestra el mensaje **No ha funcionado nada**. Luego, lo que hace este método es crear una promesa rechazada.

Otro método interesante es **Promise.all**. Este método devuelve una promesa cumplida si todas las promesas, de la colección que recibe como parámetro, son cumplidas. Si alguna se rechaza, entonces, el método devuelve una promesa rechazada. Ejemplo:

```
let promesa1=Promise.resolve("Estoy resuelta");  
let promesa2=new Promise((resolver)=>{  
  setTimeout(()=>{resolver("Resuelvo en 3s")},3000);  
});
```



```
let promesa3=new Promise((resolver)=>{
setTimeout(()=>{resolver("Resuelvo en 6s")},6000);
});
let promesaConjunta=Promise.all([promesa1,promesa2,promesa3]);

console.log("Empezando");
promesaConjunta.then((resultados)=>{
let n=1;
for(let resultado of resultados){
console.log(` Promesa nº ${n}: Mensaje:${resultado}`);
n++;
}
});
```

Se crean 3 promesas llamadas promesa1, promesa2 y promesa3. La primera genera una resolución positiva al instante, la siguiente tras 3 segundos y la tercera tras 6 segundos. Hay que recordar que `setTimeout` es asíncrona y realiza sus acciones en segundo plano.

La promesa conjunta, creada con el método **Promise.all**, exige el cumplimiento de las 3 promesas para ser considerada una promesa resuelta. En este caso pasarán 6 segundos antes de saber que las 3 están resueltas. Lo cual significa que este código muestra el texto **Empezando** y 6 segundos después, de golpe muestra este texto:

```
Empezando
Promesa nº 1: Mensaje:Estoy resuelta
Promesa nº 2: Mensaje:Resuelvo en 3s
Promesa nº 3: Mensaje:Resuelvo en 6s
```

Hasta que no se resuelve la tercera, no podremos saber si las anteriores se han resuelto. Esto significa que este método es fantástico para sincronizar acciones asíncronas.

Es importante resaltar que, si alguna promesa se rechazara o provocara un error, instantáneamente se generaría el rechazo, sin esperar al resto de promesas de la lista.

## 4.4. FUNCIONES ASYNC

Hay otras posibilidades en JavaScript de sincronización más avanzada. En la norma ES2017 apareció una esperada mejora, conocida como **await/async**. Excepto IE, los demás navegadores reconocen la estructura que vamos a ver a continuación.

Hay funciones especiales que podemos declarar anteponiendo la palabra **async**. Estas funciones, internamente son un objeto de tipo **AsyncFunction** que devuelven una promesa implícita. Pero lo que nos interesa en estos momentos es que en las funciones de tipo **async** podemos utilizar la palabra clave **await** para poder sincronizar varios elementos asíncronos.

Es decir, la función puede requerir terminar un proceso antes de iniciar un segundo proceso que dependa de él. Hasta ahora como mecanismos disponíamos de las funciones callback y de los métodos **then** y **catch** de las promesas. No es que ya no necesitemos estos elementos, es que ahora los podemos



integrar con un operador que aporta más legibilidad al código. El operador **await** requiere de una promesa, si se cumple, entonces la siguiente línea se ejecutará, si no, se mantiene en espera. Veamos un ejemplo:

```
let promesa1=Promise.resolve("Estoy resuelta");
let promesa2=new Promise((resolver)=>{
  setTimeout(()=>{resolver("Resuelvo en 3s")},3000);
});
let promesa3=new Promise((resolver)=>{
  setTimeout(()=>{resolver("Resuelvo en 6s")},6000);
});
```

```
async function esperarTiempos(){
  let mensaje1=await promesa1;
  console.log(mensaje1);
  let mensaje2=await promesa2;
  console.log(mensaje2);
  let mensaje3=await promesa3;
  console.log(mensaje3);
}
```

`esperarTiempos();`

Las 3 promesas iniciales son las que vimos en el apartado anterior para explicar el método **Promise.all**. La primera se resuelve al instante, la segunda tras 3 segundos y la tercera tarda 6 segundos. La función **esperarTiempos** se marca con la palabra clave **async** y eso permite que haya 3 variables (mensaje1, mensaje2, y mensaje3) que graben el resultado del resolver de las promesas, pero haciendo que se espere ese resultado antes de pasar a la siguiente línea de código. Eso provoca que el primer mensaje salga al instante, el segundo tras 3 segundos y el tercero saldrá a los 6 segundos. Los segundos no se acumulan porque las promesas se han creado previamente.

Diferente habría sido este código:

```
async function esperarTiempos(){
  let mensaje1=await Promise.resolve();
  console.log(mensaje1);
  let mensaje2=await new Promise((resolver)=>{
    setTimeout(()=>{resolver("Resuelvo en 3s")},3000);
  });
  console.log(mensaje2);
  let mensaje3=await new Promise((resolver)=>{
    setTimeout(()=>{resolver("Resuelvo en 6s")},6000);
  });
  console.log(mensaje3);
}
```

El código es similar, pero ahora cada promesa se genera tras esperar la finalización de la anterior. A la vista ocurre que el primer mensaje sale inmediatamente, el segundo tarda 3 segundos desde que se mostró el primero y para ver el tercer mensaje hay que esperar 6 segundos, es decir, el último mensaje aparece a los 9 segundos.

Lo interesante de **await** es la facilidad que aporta para la sincronización. Además, es muy versátil, no obliga a que lo que tiene a su derecha sea un objeto de tipo **Promise**, si es una expresión normal, la convierte en promesa y funciona también.

```
async function escribeYa(){
  let texto=await "ya está!";
  console.log(texto);
}
escribeYa();//Escribe: ya está!
```

Hay otros tipos de objetos con los que trabaja al estilo de las promesas. Son objetos que tiene implementados un método **then**. En estos objetos, el método **then** tiene que estar creado al estilo del método **then** de las promesas. Es decir, tiene que recibir una función callback para resolver y, opcionalmente, otra para rechazar. Se les llama a estos objetos, objetos **thenables**, simplemente son objetos con un método **then** correcto implementado.

Otra cuestión es qué pasa si alguna de las promesas (u objetos **thenables**) son rechazadas. ¿Cómo capturamos el rechazo?. Lo lógico para ello es que el rechazo signifique lanzar un error, por lo que basta con que dispongamos una estructura de tipo **try..catch**.

```
async function falla(){
  try{
    let resultado=await Promise.reject(Error("Promesa rechazada"));
  }
}
falla();//Escribe: Promesa rechazada
```

Incluso es posible usar el método **catch** obviando la estructura **try..catch**.

```
async function falla(){
  let resultado=await Promise.reject(Error("Promesa rechazada"))
    .catch(error=>{console.log(error.message)});
}
falla();//Escribe: Promesa rechazada
```