

# CONCEPTOS AVANZADOS

## OBJETIVOS

- Reconocer la importancia de la gestión de errores en las aplicaciones web
- Distinguir entre los conceptos de error, excepción y aviso
- Aplicar la estructura try..catch..finally para capturar los errores y gestionarlos adecuadamente
- Analizar el funcionamiento de los objetos de error
- Asimilar las ventajas del uso de módulos
- Aplicar la carga dinámica de módulos en aplicaciones web
- Distinguir las diferencias entre la programación síncrona y asíncrona
- Identificar el problema del exceso de llamadas a funciones callback
- Reconocer el funcionamiento de las promesas como mecanismo de control de acciones asíncronas
- Aplicar correctamente los métodos **then** y **catch** de las promesas
- Asimilar el funcionamiento de las funciones **async** y el operador **await**

## CONTENIDOS

- 9.1 JAVASCRIPT AVANZADO**
- 9.2 CONTROL DE ERRORES**
  - 9.2.1 INTRODUCCIÓN AL CONTROL DE ERRORES
  - 9.2.2 ERRORES, EXCEPCIONES Y AVISOS
  - 9.2.3 JAVASCRIPT MÁS ESTRICTO
  - 9.2.4 CREAR Y LANZAR ERRORES PROPIOS
  - 9.2.5 GESTIONAR LAS EXCEPCIONES. BLOQUE TRY...CATCH
- 9.3 MÓDULOS**
  - 9.3.1 USO DE MÓDULOS Y PAQUETES
  - 9.3.2 CARGA Y CREACIÓN DE MÓDULOS
- 9.4 PROGRAMACIÓN ASÍNCRONA**
  - 9.4.1 PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA
  - 9.4.2 CALLBACK HELL
  - 9.4.3 PROMESAS
  - 9.4.4 FUNCIONES **ASYNC**
- 9.5 PRÁCTICAS RESUELTAS**
- 9.6 RESUMEN DE LA UNIDAD**
- 9.7 TEST DE REPASO**

k\_\_\_\_\_ y

## 9.1 JAVASCRIPT AVANZADO

El lenguaje JavaScript ha mejorado enormemente su sintaxis para acomodarse a las necesidades planteadas por los desarrolladores de aplicaciones web. Esta unidad presenta algunos de los elementos más celebrados en las mejoras del lenguaje. Es una unidad teórica cuya aplicación se verá reforzada en el tema siguiente, cuando se describa la forma de realizar peticiones AJAX mediante la API Fetch.

Todo desarrollador debe de 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 aquí se presentan ya han sido adoptadas por todos ellos, salvo los que, como Internet Explorer, no siguen actualizándose.

## 9.2 CONTROL DE ERRORES

### 9.2.1 INTRODUCCIÓN AL CONTROL DE ERRORES

Sin duda una de las tareas más importantes, a la hora de programar aplicaciones, es la de solucionar los errores que van apareciendo. Todos los profesionales del mundo del desarrollo han pasado horas y horas, al menos alguna vez, tratando de solventar un error que habían detectado en el programa. Lo peor, es que hay veces que los errores se detectan mucho después porque ocurren circunstancias en el uso de la aplicación que no se habían tenido en cuenta.

La habilidad de encontrar errores se va consiguiendo a través de los años de experiencia, pero hay que conocer los aspectos y las técnicas fundamentales desde el primer momento. Los errores que ocurren en un programa pueden ser:

- **Errores al escribir código por parte del programador.** Son **errores de sintaxis**, errores por cosas como: expresiones incorrectas al escribir el código, cierres de llaves olvidados, palabras clave mal escritas, etc. 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 dos tipos:

- **Errores detectables en tiempo de escritura.** Son fallos de sintaxis evidentes que la mayoría de entornos de desarrollo (incluido **Visual Studio Code**) pueden marcar antes de probar el código. Muchas veces estos entornos lo que hacen es subrayar en rojo el código erróneo.
- **Errores de ejecución.** Son errores que solo se pueden detectar cuando el código se intenta ejecutar para probar la aplicación. Un ejemplo de error 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. Son **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. Puede ser también, que el error lo detecten los usuarios y se lo comuniquen a los desarrolladores.
- 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 acciones inesperadas que realiza el usuario y que causan un error en tiempo de ejecución. Por ejemplo, pedir al usuario un número y recibir un texto. En realidad, son errores lógicos que ocurren por no prever estas situaciones.

## 9.2.2 ERRORES, EXCEPCIONES Y AVISOS

Estas tres palabras se refieren al control de errores, pero es muy importante diferenciarlas para saber controlar y decidir debidamente cómo actuar si ocurre cualquiera de ellas.

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 o desarrolladora para que conozca la situación. Los avisos pueden ser vitales para detectar, de forma temprana, errores complejos de resolver.

## 9.2.3 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: la 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 se debe convertir a JavaScript con 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:

```
x=9;
consolé.log(x);
```

Simplemente veremos el número 9 por consola. Si viéramos los avisos, aparecería un aviso indicando que la variable *x* no se ha definido. La razón es que no hemos declarado la variable *x*, pero no se considera un error, es un **warning**. El programa funcionará porque actúa con la variable como si realmente se hubiera declarado.

El modo estricto se activa escribiendo el término entrecomillado '**use strict**' en la primera línea del archivo. En ese caso, el código siguiente:

```
'use strict';
x=9;
consolé.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 es el caso del ejemplo anterior, pero se puede activar para el código interior a funciones concretas:

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

La función *f* usará el modo estricto en su código, fuera de la función no se usará este modo.

## 9.2.4 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 seis objetos que sirven para crear errores propios de tipo más específico.

TIPO DE ERROR	USO
<b>EvalError</b>	Error al intentar usar la función <b>eval()</b> de JavaScript.
<b>RangeError</b>	Error numérico 0 de rango de valores incorrecto.
<b>ReferenceError</b>	Referencia a objeto no válido.
<b>TypeError</b>	Error por un mal uso de los tipo de datos .
<b>URI Error</b>	Error al codificar 0 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.

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

JavaScript aporta una estructura llamada **try..catch**. Esta estructura trabaja con esta sintaxis:

```
try{
  ...
  código que puede provocar un error
  ...
}catch(objetoError){
  ...
  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 este sencillo ejemplo:

```
try{
  consolé.log(e);
  consolé.log("aquí");
}
```

```
catch(error){
    let e=1;
    consolé.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 simplemente, 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 pantalla:

```
12
aquí
```

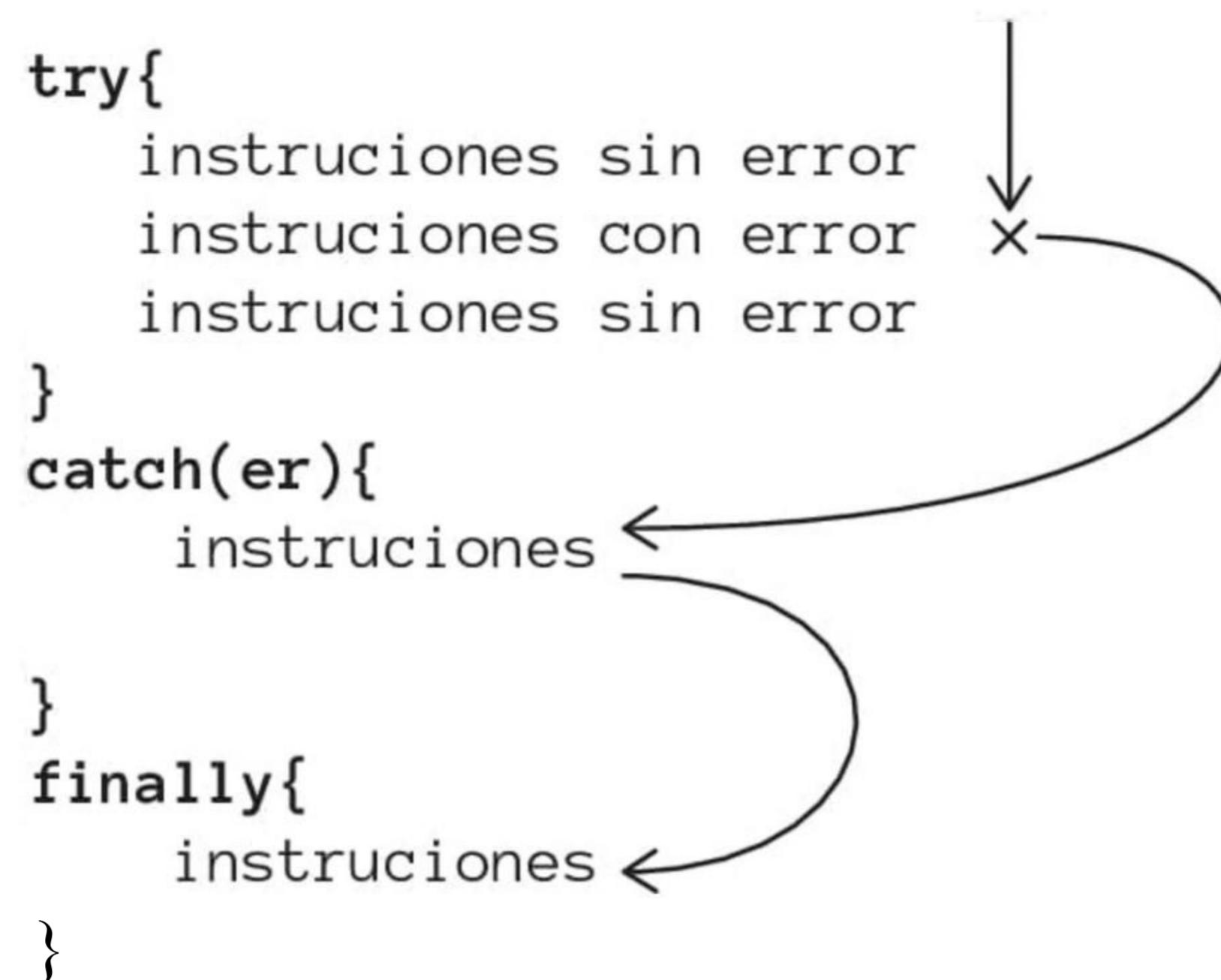
No se muestra el código del catch, no hay error.

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

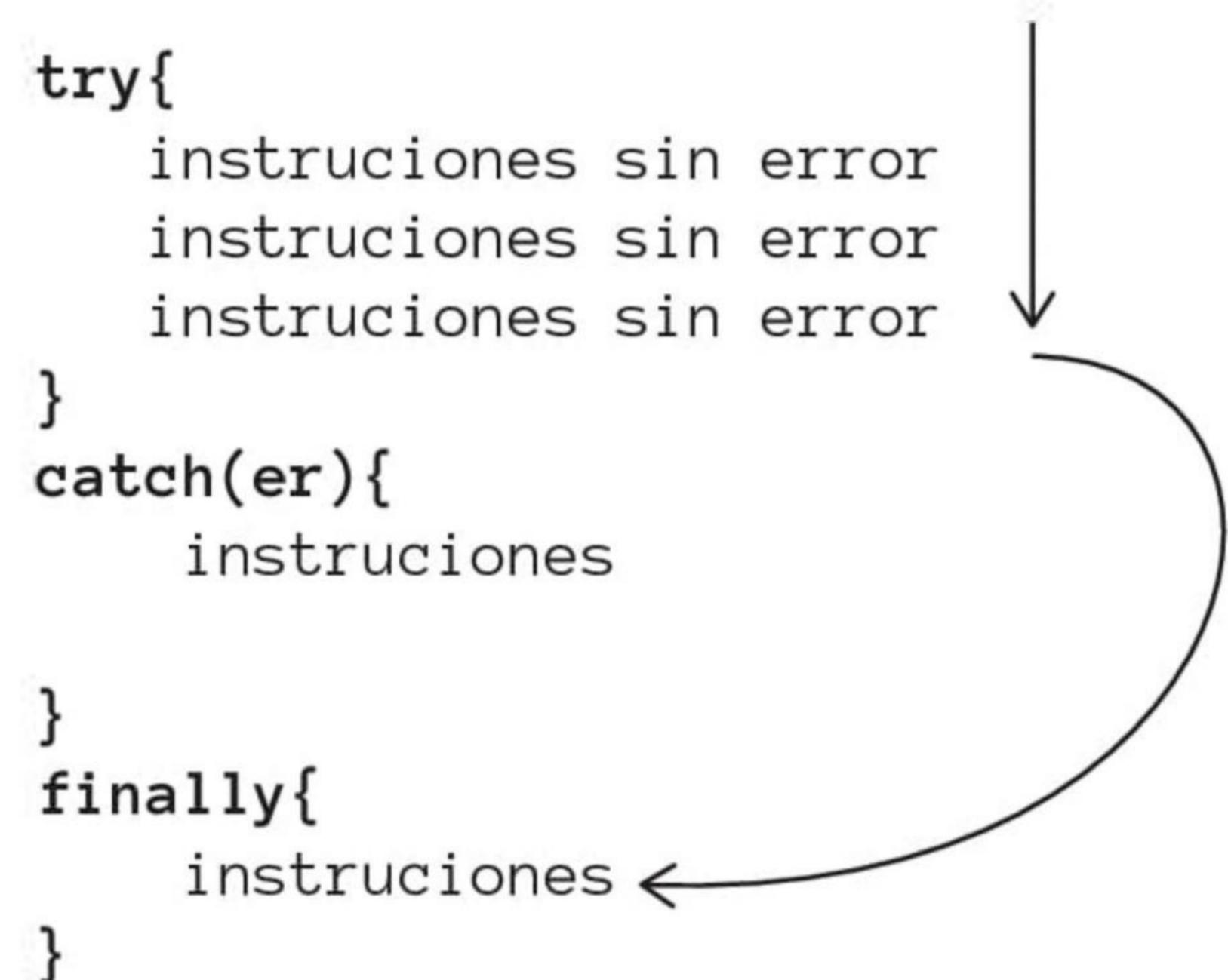
```
try{
    consolé.log(e);
    consolé.log("aquí");
}
catch(error){
    let e=1;
    consolé.log(e);
}
finally{
    consolé.log("Yo salgo siempre");
}
```

Muestra:

```
1
Yo salgo siempre
```



**Figura 9.1:** Flujo de instrucciones en una estructura try..catch..finally con un error capturado.



**Figura 9.2:** Flujo de instrucciones en una estructura try..catch..finally sin que ocurra un error

## 9.3 MÓDULOS

### 9.3.1 USO DE MÓDULOS Y PAQUETES

La programación de aplicaciones para **Node.js** ha conseguido influenciar mucho las nuevas normas sobre JavaScript. Una de sus más llamativas aportaciones al JavaScript clásico ha sido el uso de módulos. Hemos visto en temas anteriores (2.4.4 "**npm**", en la página 29 ), como node 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 ayuden en el proceso de desarrollo.

Cada paquete integra una serie de módulos. Hay que entender que un módulo es un conjunto de funciones, constantes, objetos, etc, que se usan como una librería que podremos reutilizar para facilitar el desarrollo de aplicaciones.

Sin pretender profundizar más en la creación de aplicaciones con node.js (que no son el centro del estudio de este libro), lo cierto es que los módulos permiten de forma eficaz tanto utilizar módulos de librerías de terceros, como crear módulos propios que pueden reutilizarse en todas

nuestras aplicaciones. Esto permite condensar más nuestras aplicaciones y facilitar la reusabilidad del código.

Las aplicaciones de node.js hacen uso de módulos desde hace mucho tiempo. Pero la cuestión es ¿qué pasa con el JavaScript creado para ser ejecutado en un navegador? Es decir ¿qué pasa con el JavaScript del lado del cliente?

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

```
<script src="1ibrerial.js"></script>
<script src="1ibreria2.js"> </script>
<script src="1ibreria3.js"> </script>
...
<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 carga archivos externos, genera una petición http por cada librería y eso puede ralentizar la ejecución de la aplicación.

Especialmente preocupante es el hecho de que, de algunas librerías estándar, solo usemos ciertas funcionalidades, ya que la etiqueta script carga el archivo entero. Y esto no es nada eficiente.

En definitiva, 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 del cliente.

En atención a ello han aparecido numerosas utilidades para añadir algún mecanismo de carga de módulos para el JavaScript del navegador. Ejemplos de ello son, la librería **Require.js**<sup>1</sup> y también **System.js**<sup>2</sup> que requieren de la carga 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 usar 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 los módulos cargados. Ejemplos de esta técnica son las utilidades **Browserify** o **webpack**.

Pero la carencia en el estándar seguía y de hecho, en el momento de escribir estas líneas, sigue habiendo problemas con la carga de módulos usando el estándar. Por ello, las opciones comentadas antes se siguen usando en muchos casos. Si en este libro hemos optado por comentar la carga estándar de módulos es porque parece claro que en el futuro será una práctica habitual a la hora de programar aplicaciones web.

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

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

## 9.3.2 CARGA Y CREACIÓN DE MÓDULOS

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

### 9.3.2.1 CREACIÓN DE MÓDULOS

Un módulo no es más que un conjunto de funciones, variables, objetos y todo tipo de elementos que puedan ser reutilizados en otro código. Cuando deseemos en nuestro código alguno de los elementos del módulo bastará con indicar qué queremos importar el módulo y qué elementos concretos deseamos de él (o bien importar todo).

Por otro lado, cuando se crea el 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 * pradio;
}
```

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 PI_CUADRADO=Math.PI*Math.PI;

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

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

```
const PI_CUADRADO=Math.PI*Math.PI;

function areaCirculo(radio){
    return radio * PI.CUADRADO;
}
function areaCuadrado(lado){
    return lado ** 2;
}
export{
    PI.CUADRADO,
    areaCirculo,
    areaCuadrado
}
```

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

### 93.2.2 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 "./geometría.js";
consolé.log(areaCirculo(5));
```

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

```
import { areaCirculo, PI.CUADRADO } from /geometría.js";
```

Podemos renombrar los elementos importados:

```
import { areaCirculo as circulo, areaCuadrado as cuadrado }
from "./geometría.js";

consolé.log(circulo(9)); //Escribe 88.82643960980423
consolé.log(cuadrado(4)); //Escribe 16
```

También podemos importar todos los elementos de un módulo, pero tenemos que asignar un nombre que se utilizará que usar 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 dos módulos distintos.

```
import * as geom from /geometría.js";
consolé.log(geom.areaCirculo(9));
consolé.log(geom.areaCuadrado(4));
```

Un detalle muy 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 "./geometría.js";
consolé.log(geom.areaCirculo(9));
consolé.log(geom.areaCuadrado(4));
</script>
```

## 9.4 PROGRAMACIÓN ASÍNCRONA

### 9.4.1 PROGRAMACIÓN SÍNCRONA Y ASÍNCRONA

Las funciones de tipo callback (5.4.3 "*Funciones Callback*", en la página 183 ) son uno de los elementos del lenguaje JavaScript que ayudan a entender por qué es un lenguaje asíncrono. Pero es momento ya de dedicar un tiempo a las bases de la programación asíncrona.

La mayoría de 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 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. Que JavaScript sea asíncrono implica que una instrucción puede no haber terminado su labor cuando ya se está ejecutando la siguiente.

Pero la cuestión es por qué es interesante, incluso fundamental, que JavaScript trabaje de forma asíncrona. Un ejemplo típico es una página que necesita cargar dos elementos independientes que muestran información procedente de dos servicios en 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 en 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. Si el segundo espera al primero, la carga es más lenta y el usuario de la aplicación lo notaría. Lo ideal es que la carga sea asíncrona.

En lenguaje 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 dos procesos que consigan resultados de forma independiente. JavaScript es un lenguaje de un único hilo, no se pueden programar dos hilos. Pero las operaciones sobre la red y otras operaciones de entrada/salida (como consultar bases de datos) se lanzan de forma independiente.

Pero también este efecto tiene sus problemas. ¿Y si deseamos colorear el mapa? El problema es que, aunque tengamos el código para 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.***

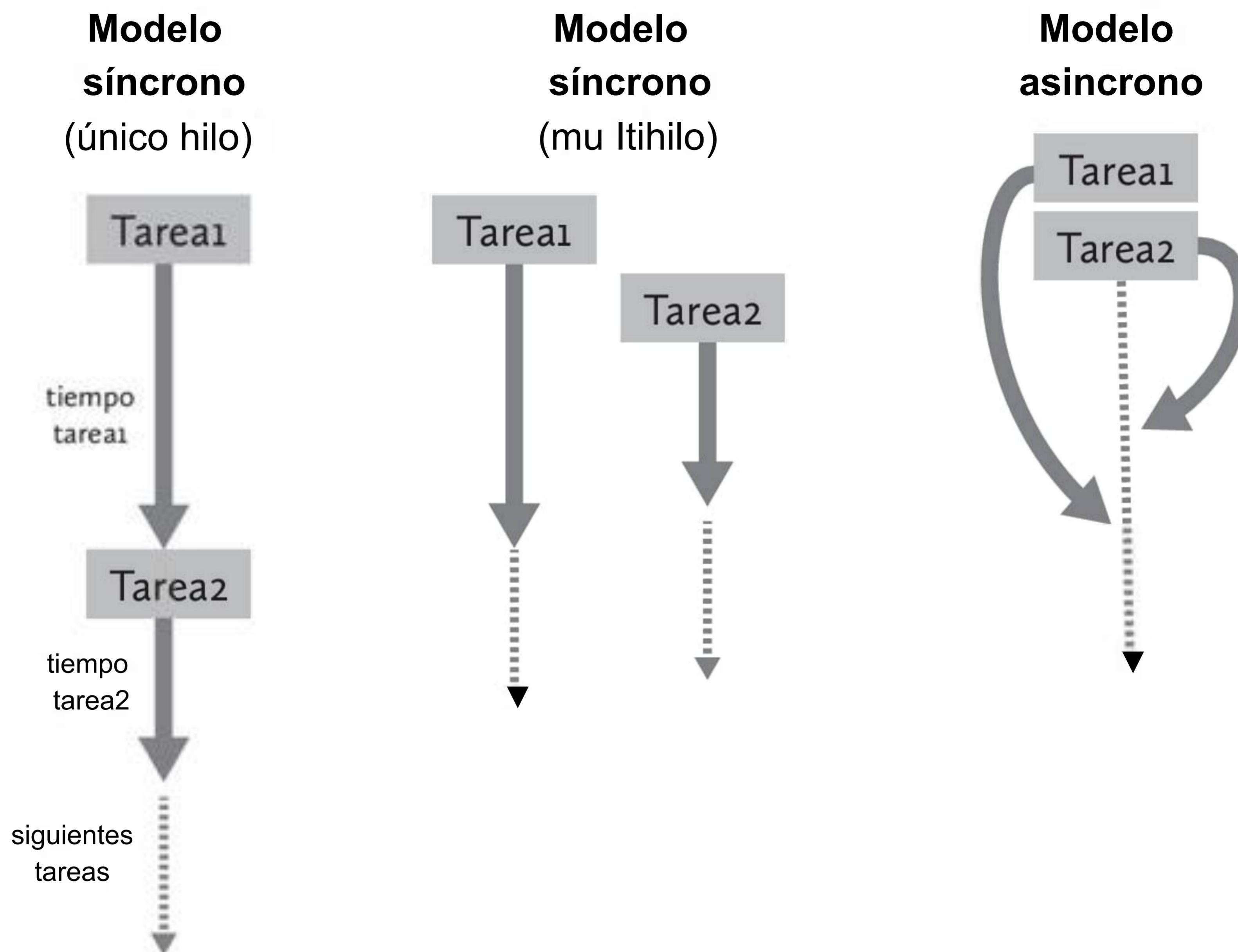


Figura 9.3: Comparación entre los modelos síncronos y asincrónos

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.

#### 9.4.2 CALLBACK HELL

Este término que podemos traducir como el 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("elick",function(ev){
    cargarMapa(componente,function(error){
        if(error){
            consolé.log("Error al cargar el mapa");
        }
    });
});
```

```

    }
    else{
        colorear(componente,function(error){
            if(error){
                consolé.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.

Ciertamente, no hay duda de que la invocación a *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, como dice el título de este apartado.

## 9.4.3 PROMESAS

### 9.4.3.1 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. Esta nueva estructura es lo que se conoce como **promesa**.

Las promesas, como tantas veces ocurre en 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) pero aportan 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.

3 <https://github.eom/kriskowal/q>

4 <http://github.com/cui0is/whe>

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

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 dos 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*)**. Independiente 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.

#### 9.43.2 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 dos parámetros, los cuales son dos funciones. La primera (se la suele llamar **resolve** o **resolver**) se invoca cuando se ha verificado que la operación ha finalizado de forma correcta. La segunda (se suele llamar **reject** o **rechazar**) se invoca cuando se ha determinado que el proceso no ha finalizado correctamente.

Ambas funciones reciben parámetros. A cada una se la 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 de promesa (la creación de la promesa) implica lanzar la tarea en segundo plano.

#### 9.4.33 MÉTODO THEN

Pero la cuestión 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.

#### 9.4.34 MÉTODO CATCH

Como ya hemos dicho, 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.

#### 9.4.33 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 consista en que, si dos variables valen lo mismo, por pantalla salga un mensaje diciéndolo, y si no así, 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)=>{
    consolé.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 **ni** 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 nl=3;
    let n2=2;
    if(nl==n2) resolver(" ;Son iguales!");
    else rechazar(Error("Algo raro ha pasado"));
});

promesa.then((respuesta)=>{
    consolé.log(respuesta);
}).catch((error)=>{
    consolé.log(error.message);
});

```

Los valores de **ni** y **n2** son distintos, lo que provoca que invoquemos a 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 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)=>{
    consolé.log("Me ha ido bien");
}).catch((error)=>{

```

```
    consolé.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){
    consolé.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.

#### 9.43.6 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){
    consolé.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 unidad siguiente cuando de verdad veamos una utilidad práctica real de las promesas y el encadenamiento de las mismas. Pero no podíamos acabar este apartado sin adelantar de manera conceptual la idea.

Volvamos al ejemplo de colorear un mapa. No podemos colorear el mapa si no hemos podido cargarlo. Supongamos, además, que debemos cargar para colorear, una plantilla de colores de Internet. Sin 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"));

```

La limpieza de este código es maravillosa. Pero la potencia la dan las funciones **cargarMapa**, **cargarPlantilla** y **colorear**. El resultado de las dos primeras tiene que ser una promesa que indique la correcta finalización de la tarea o su fallo al finalizar.

#### 9.4.37 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)=>{consolé.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)=>{consolé.log(mensaje)})
.catch((error)=>{consolé.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 promesal=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([promesal,promesa2,promesa3]);

consolé.log("Empezando");

promesaConjunta.then((resultados)=>{
    let n=1;
    for(let resultado of resultados){
        consolé.log('Promesa n2 ${n}: Mensaje:${resultado}');
        n++;
    }
});
```

Se crean tres promesas llamadas ***promesal***, ***promesa2*** y ***promesa3***. La primera genera una resolución positiva al instante, la siguiente tras tres segundos y la tercera tras seis 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 del cumplimiento de las tres promesas para ser considerada una promesa resuelta. En este caso pasarán seis segundos antes de saber que las tres están resueltas. Lo cual significa que este código muestra el texto *Empezando* y seis segundos después, de golpe, muestra este texto:

```
Empezando
Promesa n2 1: Mensaje:Estoy resuelta
Promesa n2 2: Mensaje:Resuelvo en 3s
Promesa n2 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.

## 9.4.4 FUNCIONES ASYNC

Hay otras posibilidades en JavaScript de sincronización más avanzadas. En la norma ES2017 apareció una esperada mejora, conocida como **await/async**. Actualmente, salvo Internet Explorer como es habitual, todos los navegadores actuales reconocen esta estructura que explicaremos a continuación.

Hay funciones especiales que podemos declarar anteponiendo la palabra **async**. Esto declara un tipo de función especial, que internamente es un objeto de tipo **AsyncFunction** que devuelve una promesa implícita. Pero lo que nos interesa realmente, 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 les 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;
    consolé.log(mensaje1);
    let mensaje2=await promesa2;
    consolé.log(mensaje2);
    let mensaje3=await promesa3;
    consolé.log(mensaje3);
}

esperarTiempos();
```

Las tres promesas iniciales son las que vimos en el apartado anterior para explicar el método **Promise.all**. La primera se resuelve al instante, la segunda tarda tres segundos y la tercera tarda seis segundos. La función **esperarTiempos** se marca con la palabra clave **async** y eso permite que haya tres 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 sale al instante, el segundo tres segundos desde que se inició el programa y el tercer mensaje saldrá a los seis segundos. Los segundos no se acumulan, porque las promesas se han creado previamente.

Diferente habría sido este otro código:

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

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 tres segundos desde que se mostró el primer mensaje y para ver el tercer mensaje hay que esperar seis segundos más, es decir, el último mensaje aparece a los nueve 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!";
    consolé.log(texto);
}
escribeYa(); //Escribe: ya está!
```

Hay otros tipos de objetos con los que trabaja al estilo de las promesas. Son objetos que tienen 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** correctamente 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"));
    }
    catch(error){
        consolé.log(error.message);
    }
}
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
```

## 9.5 PRÁCTICAS RESUELTA

### Práctica 9.1: Temporizador hecho con promesas

- Crear una función que reciba un número de milisegundos y genere un temporizador que escriba el texto "*Tiempo concluido*" cuando pasen dichos milisegundos.
- Realmente la función no escribe, sino que genera una promesa cuyo resultado es dicho texto.
- En el caso de que no se cumpla la promesa, el resultado será el texto "*El tiempo no va bien*".
- Se decidirá que la promesa no se ha cumplido cuando pase el doble de tiempo que el indicado en los milisegundos.

#### SOLUCIÓN: PRÁCTICA 9.1

Esta práctica busca que nos familiaricemos con la programación de promesas. Aunque, con saber cómo usarlas es suficiente para los objetivos principales de este libro, fabricar promesas nos ayudará a que esos objetivos los alcancemos de forma más óptima. Además, la creación de promesas ayuda a mejorar la forma de crear programas asíncronos.

Los temporizadores en JavaScript se crean con las funciones `setInterval` y `setTimeout` (véase "Temporizadores" en la página 279). En este caso, la idónea es `setTimeout`, porque solo necesitamos que se ejecuta una vez.

El código de la función sería el siguiente:

```
function temporizador(tiempo){
    var promesa=new Promise((resolver, rechazar)=>{
        var temp=setTimeout(()=>{
            clearTimeout(temp2);
            resolver("Tiempo concluido");
        },tiempo);
        var temp2=setTimeout(()=>{
            rechazar("El tiempo no va bien");
        },tiempo*2);
    });
    return promesa;
}
```

La función crea dos temporizadores, uno con el tiempo que recibe la función como parámetro y el segundo con el doble. El primero invoca a la función callback `resolver` porque considera cumplida la promesa. El segundo invoca a `rechazar`.

Suponiendo que estamos en una página web vacía, escribimos los resultados de la promesa como texto completo de la página. Invocaremos a la función con 500 ms.

```
temporizador(5000).then((texto)=>{
    document.body.innerHTML=
    "<p>" + texto + "</p>";
}).catch((mensaje)=>{
    document.body.textContent=mensaje;
});
```

## Práctica 9.2: Módulo de funciones de temporizadores

- Crea un archivo llamado **temporizador.js**, el cual será un módulo que podremos importar a nuestro código JavaScript principal.
- Este módulo constará de dos funciones:
  - La función **temporizador** creada en la práctica anterior.
  - La función **cuenta**. Esta función será capaz de escribir una cuenta atrás, y mostrarla en un elemento de HTML. Los parámetros son:
    - \* El número con el que se inicia la cuenta atrás.
    - \* El elemento en el que escribiremos la cuenta atrás. Por defecto usará el elemento **body**.
    - \* El intervalo en milisegundos en el que cambia cada número. Por defecto valdrá 1000.
    - \* Una función callback, cuyo código se ejecuta cuando la cuenta finalice.
- Crea, además, una aplicación web que cree en dos párrafos de modo que en el primero se cuente atrás desde el 6, pasando de segundo en segundo. En el segundo se contará desde el 60 pero cada número se moverá de décima en décima. Además, al llegar a cero, queremos que escriba "**Fin**".

---

### SOLUCIÓN: PRÁCTICA 9.2

El módulo de las dos funciones sería el siguiente:

```
/** 
 * Crea una promesa que genera un mensaje en el
 * tiempo indicado
 * @param {Number} tiempo ms
*/
export function temporizador(tiempo){
    var promesa=new Promise((resolver, rechazar)=>{
        var temp=setTimeout(()=>{
            clearTimeout(temp2);
            resolver("Tiempo concluido");
        },tiempo);
        var temp2=setTimeout(()=>{
            rechazar("El tiempo no va bien");
        },tiempo* *2);
    });
}
```

```

    return promesa;
}

/***
 * Crea una cuenta atrás en el elemento indicado
 * @param {number} numero Desde dónde contamos
 * @param {*} elemento En que elemento HTML escribimos
 * @param {*} tiempo ms en los que cambia cada número
 * @param {*} f Función opcional que queremos que se ejecute al final
*/
export      async      function      cuenta(numero,elemento=document.body,tiempo=1000,f)
{
  try{
    for(let innúmero;i>=0;i —){
      await temporizador(tiempo);
      elemento.textContent=i;
    }
    if (í) f();
  }
  catch(e){
    consolé.log("Error "+e);
  }
}

```

La función de la cuenta atrás es asíncrona, usa la función **await** y la captura de errores con **try..catch**. De esta forma, el código es más legible y usa de forma eficaz la función del temporizador creada anteriormente. Aunque lo que hace esta función es complejo, su código final resulta sencillo.

Gracias a este módulo, la página en sí resuelve el problema en apenas cuatro líneas de JavaScript:

```

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Documento</title>
  <style>
    p{
      font-size:2em;
    }
  </style>
</head>
<body>
  <p> </p>
  <p> </p>

```

```
<script type="module">
    import { temporizado!, cuenta } from " ./temporizador.js" ;
    let p1=document.querySelector("p:first-of-type");
    let p2=document.querySelector("p:last-of-type");
    cuenta(6,p1)
    cuenta(60,p2,100,()=>{
        p2.textContent="Fin";
    });
</script>
</body>
</html>
```

Las invocaciones a la función **cuenta**, hacen muy sencillo el código.

### cuenta(6,p1)

Con esta invocación, decimos que queremos contar desde el número **6** y que se muestre en el elemento **p1**. Como no indicamos tiempo, se cambiará de número en cada segundo. Al no indicar función alguna, al final no se hará nada.

```
cuenta(60,p2,100,()=>{
    p2.textContent="Fin";
});
```

Este otro código, cuenta desde el número **60**, se muestra la cuenta en el elemento **p2**, se cambiará de número cada 100 milisegundos (una décima) y al final se ejecuta el código de la función flecha que escribirá la palabra **Fin** en el propio elemento **p2**.

## 9.6 RESUMEN DE LA UNIDAD

- JavaScript ha evolucionado en estos años, incorporando nuevas estructuras en el lenguaje y mejoras que han permitido dar un nuevo impulso para manejar de forma más eficientes elementos como el control de errores, las promesas o la carga dinámica de módulos.
- El control de errores es fundamental para conseguir aplicaciones correctas y eficientes.
- Los errores se producen por distintas causas, principalmente por fallos en la programación (que se pueden descubrir de forma temprana o después de diversas pruebas) o por otro tipo de circunstancias inesperadas.
- La forma de trabajar de JavaScript no facilita el control de errores, por ello disponemos del modo estricto que se hace más sensible a los errores.
- La gestión de excepciones (errores que cortan la ejecución del programa) en JavaScript se puede gestionar con la ayuda de la estructura **try..catch**, la cual permite que si una excepción se produce, el flujo sea controlado por la estructura **catch**, la cual manejará de la mejor forma el error ocurrido.
- La instrucción **catch** recibe un objeto de error que contiene los detalles del mismo. El propio programador puede crear objetos de error e incluso lanzarlos con ayuda de la instrucción **throw**.
- La estructura **try..catch** puede incorporar un bloque **finally** cuyas instrucciones se ejecutan tanto si hay error, como si no.
- La carga de módulos permite un uso de librerías de funciones y objetos más eficientes que la carga con la etiqueta **script** de HTML.
- Podemos crear módulos utilizando la palabra clave **export** para indicar qué elementos son exportables del módulo. Al cargar el módulo, mediante la instrucción **import** podemos decidir qué elementos concretos importamos.
- JavaScript es un lenguaje asíncrono, lo que implica que hay acciones que finalizan en segundo plano. Eso es una ventaja para la programación de aplicaciones web.
- Necesitamos en muchas ocasiones poder sincronizar estas acciones. Un primer mecanismo fueron las funciones callback, pero tienden a complicarse enormemente produciendo el llamado **callback hell**, cuando encadenamos varias acciones a sincronizar.
- Las promesas permiten valorar una acción y decidir si finaliza bien o mal. Disponen de un método **then** cuyo código (envuelto en una función callback) se ejecuta en caso de que la promesa se resuelva y de un método llamado **catch**, cuyo código se ejecuta en caso de que la promesa no se resuelva.
- El uso de promesas permite el encadenamiento de las mismas encadenando métodos **then** y **catch**. Esta estructura es más legible que el uso anidado de funciones callback para controlar acciones asíncronas.

Los objetos Promise aportan métodos para generar promesas cumplidas y rechazadas, así como para crear promesas a partir del cumplimiento de una serie de varias promesas.

La norma ES2017 trajo la posibilidad de crear funciones **async**, las cuales admiten el uso del operador **await** para generar código síncrono, basado en promesas, aun más legible y eficiente.

## 9.7 TEST DE REPASO

1.- Una acción asincrona de JavaScript va a tardar 2 segundos en realizarse. Justo detrás, en el código, se lanza otra acción que tardará en completarse 1 segundo. ¿Cuánto tardan, en teoría, en JavaScript en completarse ambas acciones?

- a) 3 segundos.
- b) 2 segundos.
- c) 1 segundos.
- d) Instantáneamente.

2.- Se programan las acciones anteriores de forma síncrona. ¿Cuánto tardan en completarse ambas acciones?

- a) 3 segundos.
- b) 2 segundos.
- c) 1 segundos.
- d) Instantáneamente.

3.- Hemos creado un módulo y sabemos que está correctamente hecho. Sabemos que la instrucción de importación también. El código está dentro de una etiqueta script de HTML y se está ejecutando en la última versión de Mozilla Firefox. ¿Qué puede pasar?

- a) Que solo funciona la importación si el código está en un archivo externo.
- b) Que Firefox no acepta la importación dinámica de módulos.
- c) Que no hemos usado type="module" en la etiqueta script.
- d) Las dos últimas son correctas.

4.- Queremos importar las funciones colorear y recortar del archivo paint.js. ¿Qué código lo hace posible?

- a) `import colorear,recortar from paint.js`
- b) `import colorear,recortar from "./paint.js"`
- c) `import { colorear,recortar } from "./paint.js"`
- d) Todas son correctas

5.- ¿Qué diferencia hay entre error y excepción?

- a) La excepción es controlable, el error no.
- b) El error es controlable, la excepción no.
- c) Ninguna.

6.- En qué norma del estándar ECMAScript apareció la estructura `async/await`

- a) ES2015
- b) ES2016
- c) ES2017
- d) ES2018

7.- En qué norma del estándar ECMAScript aparecieron las promesas

- a) ES2015
- b) ES2016
- c) ES2017
- d) ES2018

8.- ¿Qué escribe este código?

`try{`

```
    consolé.log("punto 1");
    throw new Error("mi error");
    consolé.log("punto 2");
}
```

`catch(error){`

```
    consolé.log("punto 3");
}
```

- a) punto 1  
punto 2  
punto 3
- b) punto 2  
punto 3
- c) punto 1  
punto 3
- d) punto 1  
punto 2
- e) punto 3
- f) punto 1

9.- ¿Qué escribe este código?

```
try{
    consolé.log("punto 1");
    throw new Error("mi error");
    consolé.log("punto 2");
}
catch(error){
    consolé.log("punto 3");
}
finally{
    consolé.log("punto 4");
}
```

- a) punto 1  
punto 2  
punto 3  
punto 4
- b) punto 2  
punto 3  
punto 4
- c) punto 1  
punto 2  
punto 3
- d) punto 1  
punto 3
- e) punto 1  
punto 4
- f) punto 1  
punto 3  
punto 4

10.- Hemos visto que `Promise.all` recibe una lista de promesas y devuelve una nueva promesa. ¿Cuándo se cumple la promesa?

- a) Cuando se cumple cualquier promesa de la lista
- b) Cuando se cumplen todas las promesas de la lista
- c) Cuando se cumplen o se rechazan todas las promesas de la lista
- d) Cuando se cumple o se rechaza al menos una promesa de la lista

11.- ¿Qué escribe este código?

```
Promise.resolve(3!=3)
    .then((o)=>consolé.log(1))
    .catch((o) = >consolé.log(2));
```

- a) 3!=3
- b) 1
- c) 2
- d) Nada, hay un error.

12.- ¿Qué escribe este código?

```
async function f(){
    try{
        let b=await Promise.
            resolve(3!=3);
        consolé.log(1);
    }catch(e){
        consolé.log(2)
    }
}
f();
```

- a) 3!=3
- b) 1
- c) 2
- d) Nada, se queda esperando indefinidamente

13.- ¿Qué escribe este código

```
new Promise((a,b)=>{
    if(2==2) b("ok");
    else a("fallo");
}).then((o)=>{
    consolé.log(o);
}).catch((o)=>{
    consolé.log(o);
});
```

- a) ok
- b) fallo
- c) Nada, ocurre un error