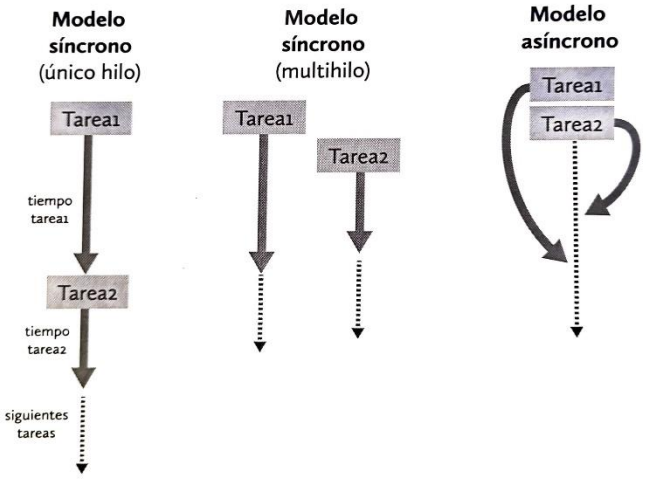
Contenido

Promesas 1
Funciones ASYNC 4

ASINCRONÍA

JavaScript es un lenguaje **asíncrono** → Una función puede no haber terminado su labor cuando ya empieza la siguiente.



Imaginad una página que carga información de dos servicios de Internet, en una capa muestra la temperatura prevista para hoy, y otra capa muestra como llegar a nuestra oficina. En programación síncrona si no se carga la temperatura no se carga el mapa. Los lenguajes síncronos solucionan el problema creando **diferentes hilos**.

La **programación asíncrona también 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 de algún tipo; **cuando llegue el mapa coloreamos**

Solución a esto ya tenemos:

- Funciones callback → Callback Hell, tener que sincronizar muchas acciones provoca un exceso de funciones callback que van desplazando el código a su derecha haciéndolo poco legible.
- Promesas→Nueva estructura que soluciona los problemas anteriores
- Funciones Async

Promesas

- Son objetos que se construyen indicando una función del tipo callback
- Esta función acepta dos parámetros, los cuales son dos funciones, la primera se 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 y se invoca cuando el proceso no ha finalizado correctamente. Para la función de rechazo el parámetro suele ser un objeto de error, ya que se permite la gestión de errores de forma más eficiente.

- ¿Cuándo recogemos los resultados del éxito o fracaso? Ahí interviene el método **then.** Este método acepta una función callback, que será invoada cuando la tarea de la promesa finalice con éxito. Hay un segundo parámetro opcional que es una función que se invoca si el resultado es erróneo. Sintaxis:

```
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 punto anterior, durante la creación de la promesa. La segunda función recibe en parámetro indicado en la función **rechazar.** Es poco habitual usar la segunda función, en su lugar se utiliza el formato de captura de errores que veremos a continuación.

- Método **catch,** nos permite gestionar el rechazo de la promesa. Es un formato 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){

…};
```

Ejemplo1:

```
let promesa=new Promise((resolver,rechazar)=>{

    let n1=3;
    let n2=3;
    if(n1==n2) resolver("Son iguales");
    else rechazar (Error("No son iguales, algo raro ha pasado"))
});

promesa.then(function(respuesta){
    console.log(respuesta);},function(error){console.log("Error"+error);});
```

Ejemplo2:

```
let promesa=new Promise((resolver,rechazar)=>{

      let n1=3;
      let n2=5;
      if(n1==n2) resolver("Son iguales");
      else rechazar (Error("No son iguales, algo raro ha pasado"))
});

promesa.then((respuesta)=>{
    console.log(respuesta);
}).catch((error)=>{console.log("Error"+error)});
```

- **Encadenamiento de métodos.** Se pueden invocar varias veces a las funciones de resolución y rechazo, pero cada método **then** o **catch** solo puede responder a una. Lo que si se puede es encadenar los métodos **then** o **catch** las veces que haga falta. Es una buena forma de estructurar nuestras funciones evitando un exceso de control con funciones callbak (**callback hell**).

Ejemplo 3:

```
let promesa = new Promise((resolver,rechazar)=>{
    let n=0;
    let intervalo=setInterval(()=>{
        n++;
        if(n==5){
            resolver("Han pasado 5 segundos");
            clearInterval(intervalo);
        }
    },1000);
});

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

A los 5 segundos se muestra hemos llegado a 5 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.

Volviendo al ejemplo colorear, la idea con promesad sería

```
cargarMapa()
        .then(mapa)=>cargarPlantillaMapa(mapa))
        .then(mapla)=>colorear(mapa);
        .catch(throw new error(“Error en la carga”));
```

- **Métodos del objeto promise:**
    - **Promise.resolve** → Crea una nueva promesa cumplida

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

    - **Promise.reject** → Crea una promesa rechazada

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

- **Promise.all** → Resuelve una promesa cumplida si todas las promesas que recibe como parámetro son cumplidas, si alguna se rechaza, entonces, el método devuelve una promesa rechazada.

  Ejemplo 4:

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

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

  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.

Funciones ASYNC

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 clava **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 de sincronización disponíamos de las funciones **callback** y de los métodos **then** y **catch** de las promesas. Veamos un ejemplo:

Ejemplo 5

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

La función **esperarTiempos** se marca con la palabra **async** y eso permite que haya tres variables (mensaje1, mensaje2 y mensaje3) que graben el resultado de resolver las promesas pero haciendo que se espere ese resultado antes de pasar a la siguiente línea de código. Los segundos no se acumulan porque las promesas se han creado previamente.

Diferente es el **ejemplo 6**

```
async function esperarTiempos(){
    let mensaje1=await Promise.resolve("Estoy resuelta");
    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);
}

esperarTiempos();
```

El código es similar, pero ahora cada promesa se genera tras esperar la finalización de la anterior.

Otra cuestión es ¿qué pasa si alguna de las promesas es rechazada?, ¿Cómo capturamos el rechazo? Lo lógico para ello es que el rechazo signifique lanzar un error, por lo que basta con utilizar la estructura **try..catch**

Ejemplo 7

```
async function falla(){
    try{
        let resultado=await Promise.reject("Estoy rechazada");

    }catch(error){
        console.log(error);
    }

}

falla();
```