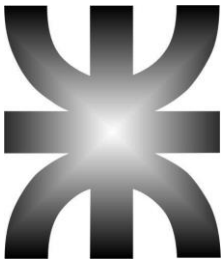


# PROMISIS - PROMESAS

## UNIDAD #5 - JS

*TUP - Laboratorio de Computación II*

*MATERIAL COMPLEMENTARIO*

## Promesa

*Imagina que sos un gran cantante y los fanáticos te preguntan día y noche por tu próxima canción.*

*Para obtener algo de alivio, prometes enviárselos cuando se publique. Le das a tus fans una lista. Ellos pueden registrar allí sus direcciones de correo electrónico, de modo que cuando la canción esté disponible, todas las partes suscritas la reciban instantáneamente. E incluso si algo sale muy mal, digamos, un incendio en el estudio tal que no puedas publicar la canción, aún se les notificará.*

*Todos están felices: tú, porque la gente ya no te abruma, y los fanáticos, porque no se perderán la canción.*

Esta es una analogía de la vida real para las cosas que a menudo tenemos en la programación:

1. Un "código productor" que hace algo y toma tiempo. Por ejemplo, algún código que carga los datos a través de una red. Eso es un "cantante".
2. Un "código consumidor" que quiere el resultado del "código productor" una vez que está listo. Muchas funciones pueden necesitar ese resultado. Estos son los "fans".
3. Una *promesa* es un objeto JavaScript especial que une el "código productor" y el "código consumidor". En términos de nuestra analogía: esta es la "lista de suscripción". El "código productor" toma el tiempo que sea necesario para producir el resultado prometido, y la "promesa" hace que ese resultado esté disponible para todo el código suscrito cuando esté listo.

La analogía no es terriblemente precisa, porque las promesas de JavaScript son más complejas que una simple lista de suscripción: tienen características y limitaciones adicionales. Pero está bien para empezar.

La sintaxis del constructor para un objeto promesa es:

```
let promise = new Promise(function (resolve, reject) {
    // Ejecutor (el código productor, "cantante")
```

```
});
```

La función pasada a `new Promise` se llama *ejecutor*. Cuando se crea `new Promise`, el ejecutor corre automáticamente. Este contiene el código productor que a la larga debería producir el resultado. En términos de la analogía anterior: el ejecutor es el "cantante".

Sus argumentos `resolve` y `reject` son callbacks proporcionadas por el propio JavaScript. Nuestro código solo está dentro del ejecutor.

Cuando el ejecutor, más tarde o más temprano, eso no importa, obtiene el resultado, debe llamar a una de estos callbacks:

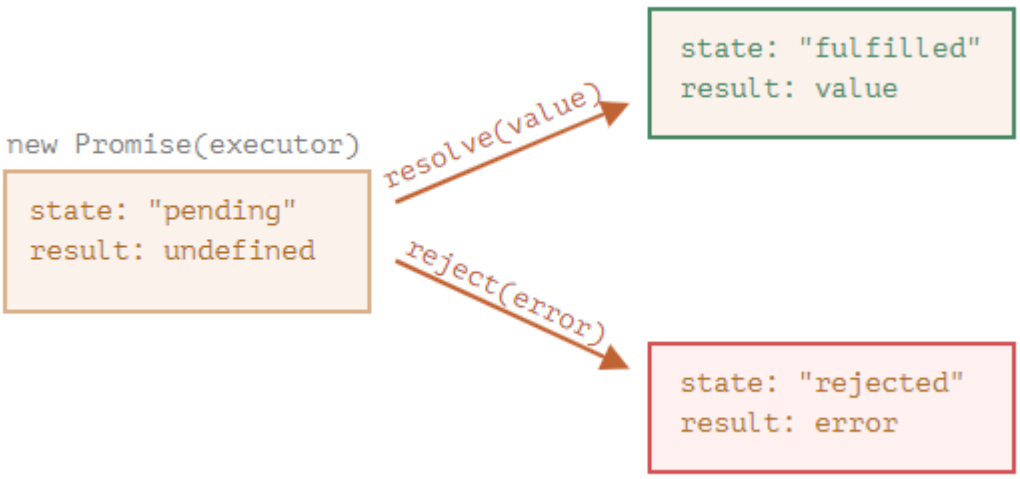
- `resolve(value)` – si el trabajo finalizó con éxito, con el resultado `value`.
- `reject(error)` – si ocurrió un error, `error` es el objeto error.

Para resumir: el ejecutor corre automáticamente e intenta realizar una tarea. Cuando termina con el intento, llama a `resolve` si fue exitoso o `reject` si hubo un error.

El objeto `promise` devuelto por el constructor `new Promise` tiene estas propiedades internas:

- `state` – inicialmente `"pendiente"`; luego cambia a `"cumplido"` cuando se llama a `resolve`, o a `"rechazado"` cuando se llama a `reject`.
- `result` – inicialmente `undefined`; luego cambia a `valor` cuando se llama a `resolve(valor)`, o a `error` cuando se llama a `reject(error)`.

Entonces el ejecutor, en algún momento, pasa la `promise` a uno de estos estados:




Después veremos cómo los "fanáticos" pueden suscribirse a estos cambios.

Aquí hay un ejemplo de un constructor de promesas y una función ejecutora simple con "código productor" que toma tiempo (a través de `setTimeout`):

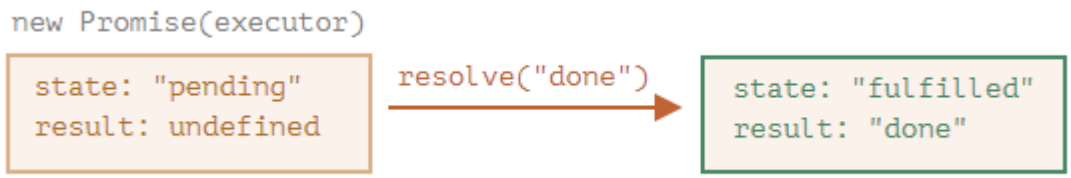
```
let promise = new Promise(function (resolve, reject) {
    // la función se ejecuta automáticamente cuando se construye la promesa

    // después de 1 segundo, indica que la tarea está hecha con el resultado
"hecho"
    setTimeout(() => resolve("hecho"), 1000);
});
```

Podemos ver dos cosas al ejecutar el código anterior:

1. Se llama al ejecutor de forma automática e inmediata (por `new Promise`).
2. El ejecutor recibe dos argumentos: `resolve` y `reject`. Estas funciones están predefinidas por el motor de JavaScript, por lo que no necesitamos crearlas. Solo debemos llamar a uno de ellos cuando esté listo.

   Después de un segundo de "procesamiento", el ejecutor llama a `resolve("hecho")` para producir el resultado. Esto cambia el estado del objeto `promise`:

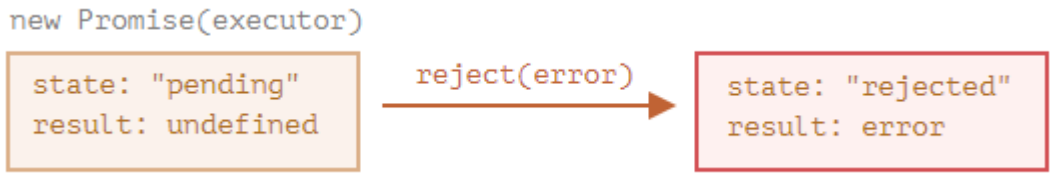



Ese fue un ejemplo de finalización exitosa de la tarea, una "promesa cumplida".

Y ahora un ejemplo del ejecutor rechazando la promesa con un error:

```
let promise = new Promise(function (resolve, reject) {
    // después de 1 segundo, indica que la tarea ha finalizado con un error
    setTimeout(() => reject(new Error("¡Vaya!")), 1000);
});
```

La llamada a `reject(...)` mueve el objeto promise al estado `"rechazado"`:




Para resumir, el ejecutor debe realizar una tarea (generalmente algo que toma tiempo) y luego llamar a "resolve" o "reject" para cambiar el estado del objeto promise correspondiente.

Una promesa que se resuelve o se rechaza se denomina "resuelta", en oposición a una promesa inicialmente "pendiente".

**Solo puede haber un único resultado, o un error**

El ejecutor debe llamar solo a un 'resolve' o un 'reject'. Cualquier cambio de estado es definitivo.

Se ignoran todas las llamadas adicionales de 'resolve' y 'reject':

```
let promise = new Promise(function (resolve, reject) {
    resolve("hecho");

    reject(new Error("…")); // ignorado
    setTimeout(() => resolve("…")); // ignorado
});
```

La idea es que una tarea realizada por el ejecutor puede tener solo un resultado o un error.

Además, `resolve/reject` espera solo un argumento (o ninguno) e ignorará argumentos adicionales.

**Rechazar con objetos `Error`**

En caso de que algo salga mal, el ejecutor debe llamar a 'reject'. Eso se puede hacer con cualquier tipo de argumento (al igual que `resolve`). Pero se recomienda usar objetos `Error` (u objetos que hereden de `Error`). El razonamiento para eso pronto se hará evidente.

**Inmediatamente llamando a `resolve/reject`**

En la práctica, un ejecutor generalmente hace algo de forma asíncrona y llama a `resolve/reject` después de un tiempo, pero no está obligado a hacerlo así. También podemos llamar a `resolve` o `reject` inmediatamente:

```
let promise = new Promise(function (resolve, reject) {
    // sin que nos quite tiempo para hacer la tarea
    resolve(123); // dar inmediatamente el resultado: 123
});
```

Por ejemplo, esto puede suceder cuando comenzamos una tarea, pero luego vemos que todo ya se ha completado y almacenado en caché.

Está bien. Inmediatamente tenemos una promesa resuelta.

### `state` y `result` son internos

Las propiedades `state` y `result` del objeto Promise son internas. No podemos acceder directamente a ellas. Podemos usar los métodos `.then`/`.catch`/`.finally` para eso. Se describen a continuación.

## Consumidores: then y catch

Un objeto Promise sirve como enlace entre el ejecutor (el "código productor" o el "cantante") y las funciones consumidoras (los "fanáticos"), que recibirán un resultado o un error. Las funciones de consumo pueden registrarse (suscribirse) utilizando los métodos `.then` y `.catch`.

# then

El más importante y fundamental es `.then`.

La sintaxis es:

```
promise.then(
    function (result) { /* manejar un resultado exitoso */ },
    function (error) { /* manejar un error */ }
);
```

El primer argumento de `.then` es una función que se ejecuta cuando se resuelve la promesa y recibe el resultado.

El segundo argumento de `.then` es una función que se ejecuta cuando se rechaza la promesa y recibe el error.

Por ejemplo, aquí hay una reacción a una promesa resuelta con éxito:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve("hecho!"), 1000);
});

// resolve ejecuta la primera función en .then
promise.then(
    result => alert(result), // muestra "hecho!" después de 1 segundo
    error => alert(error) // no se ejecuta
);
```

La primera función fue ejecutada.

Y en el caso de un rechazo, el segundo:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// reject ejecuta la segunda función en .then
promise.then(
    result => alert(result), // no se ejecuta
    error => alert(error) // muestra "Error: ¡Vaya!" después de 1 segundo
);
```

Si solo nos interesan las terminaciones exitosas, entonces podemos proporcionar solo un argumento de función para `.then`:

```
let promise = new Promise(resolve => {
    setTimeout(() => resolve("hecho!"), 1000);
});

promise.then(alert); // muestra "hecho!" después de 1 segundo
```

## catch

Si solo nos interesan los errores, entonces podemos usar `null` como primer argumento: `.then(null, errorHandlingFunction)`. O podemos usar `.catch(errorHandlingFunction)`, que es exactamente lo mismo:

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// .catch(f) es lo mismo que promise.then(null, f)
promise.catch(alert); // muestra "Error: ¡Vaya!" después de 1 segundo
```

La llamada `.catch(f)` es un análogo completo de `.then(null, f)`, es solo una abreviatura.

### Limpieza: finally

Al igual que hay una cláusula `finally` en un `try {...} catch {...}` normal, hay un `finally` en las promesas.

La llamada `.finally(f)` es similar a `.then(f, f)` en el sentido de que `f` siempre se ejecuta cuando se resuelve la promesa: ya sea que se resuelva o rechace.

La idea de `finally` es establecer un manejador para realizar la limpieza y finalización después de que las operaciones se hubieran completado.

Por ejemplo, detener indicadores de carga, cerrar conexiones que ya no son necesarias, etc.

Puedes pensarlo como el finalizador de la fiesta. No importa si la fiesta fue buena o mala ni cuántos invitados hubo, aún necesitamos (o al menos deberíamos) hacer la limpieza después.

El código puede verse como esto:

```
new Promise((resolve, reject) => {
    /* hacer algo para tomar tiempo y luego llamar a resolve o reject */
})
    // se ejecuta cuando se cumple la promesa, no importa con éxito o no
    .finally(() => stop loading indicator)
    // así el indicador de carga siempre es detenido antes de que sigamos
adelante
    .then(result => show result, err => show error)
```

Sin embargo, note que `finally(f)` `no es exactamente un alias de`then(f, f)`.

Hay diferencias importantes: `

1. Un manejador `finally` no tiene argumentos. En `finally` no sabemos si la promesa es exitosa o no. Eso está bien, ya que usualmente nuestra tarea es realizar procedimientos de finalización “generales”.

   Por favor observe el ejemplo anterior: como puede ver, el manejador de `finally` no tiene argumentos, y lo que sale de la promesa es manejado en el siguiente manejador.

2. Resultados y errores pasan “a través” del manejador de `finally`. Estos pasan al siguiente manejador que se adecúe.

   Por ejemplo, aquí el resultado se pasa a través de `finally` a `then`:

```
new Promise((resolve, reject) => {
    setTimeout(() => resolve("valor"), 2000)
})
    .finally(() => alert("Promesa lista")) // se dispara primero
    .then(result => alert(result)); // <-- .luego muestra "valor"
```

Como puede ver, el "valor" devuelto por la primera promesa es pasado a través de `finally` al siguiente `then`.

Esto es muy conveniente, porque `finally` no está destinado a procesar el resultado de una promesa. Como dijimos antes, es el lugar para hacer la limpieza general sin importar cuál haya sido el resultado.

Y aquí, el ejemplo de un error para que veamos cómo se pasa, a través de `finally`, a `catch`:

```
new Promise((resolve, reject) => {
    throw new Error("error");
})
    .finally(() => alert("Promesa lista"))  // primero dispara
    .catch(err => alert(err));  // <-- .catch muestra el error
```

3. Un manejador de `finally` tampoco debería devolver nada. Y si lo hace, el valor devuelto es ignorado silenciosamente.

   La única excepción a esta regla se da cuando el manejador mismo de `finally` dispara un error. En ese caso, este error pasa al siguiente manejador de error en lugar del resultado previo al finally.

Para summarizar:

- Un manejador `finally` no obtiene lo que resultó del manejador previo (no tiene argumentos). Ese resultado es pasado a través de él al siguiente manejador.
- Si el manejador de `finally` devuelve algo, será ignorado.
- Cuando es `finally` el que dispara el error, la ejecución pasa al manejador de error más cercano.

Estas características son de ayuda y hacen que las cosas funcionen tal como corresponde si "finalizamos" con `finally` como se supone: con procedimientos de limpieza genéricos.

**Podemos adjuntar manejadores a promesas ya establecidas**

Si una promesa está pendiente, los manejadores `.then/catch/finally` esperan por su resolución.

Podría pasar a veces que, cuando agregamos un manejador, la promesa ya se encuentre establecida.

Em tal caso, estos manejadores simplemente se ejecutarán de inmediato:

```
// la promesa se resuelve inmediatamente después de la creación
```

```
let promise = new Promise(resolve => resolve("hecho!"));

promise.then(alert); // ¡hecho! (aparece ahora)
```

Ten en cuenta que esto es diferente y más poderoso que el escenario de la "lista de suscripción" de la vida real. Si el cantante ya lanzó su canción y luego una persona se registra en la lista de suscripción, probablemente no recibirá esa canción. Las suscripciones en la vida real deben hacerse antes del evento.

Las promesas son más flexibles. Podemos agregar manejadores en cualquier momento: si el resultado ya está allí, nuestros manejadores lo obtienen de inmediato.

## Ejemplo: loadScript

A continuación, veamos ejemplos más prácticos de cómo las promesas pueden ayudarnos a escribir código asincrónico.

Tenemos, del capítulo anterior, la función `loadScript` para cargar un script.

Aquí está la variante basada callback, solo para recordarnos:

```
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => callback(null, script);
    script.onerror = () => callback(new Error(`Error de carga de script para $
{src}`));

    document.head.append(script);
}
```

Reescribámoslo usando Promesas.

La nueva función `loadScript` no requerirá una callback. En su lugar, creará y devolverá un objeto Promise que se resuelve cuando se completa la carga. El código externo puede agregar manejadores (funciones de suscripción) usando `.then`:

```
function loadScript(src) {
    return new Promise(function (resolve, reject) {
        let script = document.createElement('script');
        script.src = src;

        script.onload = () => resolve(script);
        script.onerror = () => reject(new Error(`Error de carga de script para
$ {src}`));
```

```
        document.head.append(script);
    });
}
```

Uso:

```
let promise =
loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js
");

promise.then(
    script => alert(`${script.src} está cargado!`),
    error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('Otro manejador...'));
```

Podemos ver inmediatamente algunos beneficios sobre el patrón basado en callback:

| Promesas | Callbacks |
|---|---|
| **Las promesas nos permiten hacer las cosas en el orden natural. Primero, ejecutamos `loadScript (script)`, y `.then` escribimos qué hacer con el resultado.** | Debemos tener una función `callback` a nuestra disposición al llamar a 'loadScript(script, callback)'. En otras palabras, debemos saber qué hacer con el resultado *antes* de llamar a `loadScript`. |
| **Podemos llamar a ".then" en una promesa tantas veces como queramos. Cada vez, estamos agregando un nuevo "fan", una nueva función de suscripción, a la "lista de suscripción". Más sobre esto en el próximo capítulo: Encadenamiento de promesas.** | Solo puede haber un callback. |

# Casos

## ¿Volver a resolver una promesa?

¿Cuál es el resultado del código a continuación?

```
let promise = new Promise(function (resolve, reject) {
    resolve(1);

    setTimeout(() => resolve(2), 1000);
});

promise.then(alert);
```

Resultado:

La salida es: `1`.

La segunda llamada a 'resolve' se ignora, porque solo se tiene en cuenta la primera llamada de 'reject/resolve'. Otras llamadas son ignoradas.

## Demora con una promesa

La función incorporada `setTimeout` utiliza callbacks. Crea una alternativa basada en promesas.

La función `delay(ms)` debería devolver una promesa. Esa promesa debería resolverse después de `ms` milisegundos, para que podamos agregarle `.then`, así:

```
function delay(ms) {
    // tu código
}

delay(3000).then(() => alert('se ejecuta después de 3 segundos'));
```

Solución:

```
function delay(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

## Círculo animado con promesa

Vuelva a escribir la función `showCircle` en la solución de la tarea Círculo animado con función de callback para que devuelva una promesa en lugar de aceptar un callback.

Nueva forma de uso:

```
showCircle(150, 150, 100).then(div => {
    div.classList.add('message-ball');
    div.append("Hola, mundo!");
});
```

Solucion

```
<!DOCTYPE html>
```

```html
<html>

<head>
    <meta charset="utf-8">
    <style>
        .message-ball {
            font-size: 20px;
            line-height: 200px;
            text-align: center;
        }

        .circle {
            transition-property: width, height;
            transition-duration: 2s;
            position: fixed;
            transform: translateX(-50%) translateY(-50%);
            background-color: red;
            border-radius: 50%;
        }
    </style>
</head>

<body>

    <button onclick="go()">Haz clic</button>

    <script>

        function go() {
            showCircle(150, 150, 100).then(div => {
                div.classList.add('message-ball');
                div.append("Hola, mundo!");
                div.style.backgroundColor = 'blue';
            });
        }

        function showCircle(cx, cy, radius) {
            let div = document.createElement('div');
            div.style.width = 0;
            div.style.height = 0;
            div.style.left = cx + 'px';
            div.style.top = cy + 'px';
            div.className = 'circle';
            document.body.append(div);

            return new Promise(resolve => {
                setTimeout(() => {
                    div.style.width = radius * 2 + 'px';
```

```
                    div.style.height = radius * 2 + 'px';

                    div.addEventListener('transitionend', function handler() {
                        div.removeEventListener('transitionend', handler);
                        resolve(div);
                    });
                }, 1000);
            })
        }
    </script>


</body>

</html>
```

## Encadenamiento

Una necesidad común es el ejecutar dos o más operaciones asíncronas seguidas, donde cada operación posterior se inicia cuando la operación previa tiene éxito, con el resultado del paso previo. Logramos esto creando una cadena de objetos `promises`.

Aquí está la magia: la función `then()` devuelve una promesa nueva, diferente de la original:

```
const promesa = hazAlgo();
const promesa2 = promesa.then(exitoCallback, falloCallback);
```

o

```
let promesa2 = hazAlgo().then(exitoCallback, falloCallback);
```

Esta segunda promesa (`promesa2`) representa no sólo la terminación de `hazAlgo()`, sino también de `exitoCallback` o `falloCallback` que pasaste, las cuales pueden ser otras funciones asíncronas devolviendo una promesa. Cuando ese es el caso, cualquier función callback añadida a `promesa2` se queda en cola detrás de la promesa devuelta por `exitoCallback` o `falloCallback`.

Básicamente, cada promesa representa la terminación de otro paso (asíncrono on no) en la cadena.

En el pasado, hacer varias operaciones asíncronas en fila conduciría a la clásica pirámide de funciones callback:

```
hazAlgo(function(resultado) {
    hazAlgoMas(resultado, function(nuevoResultado) {
      hazLaTerceraCosa(nuevoResultado, function(resultadoFinal) {
        console.log('Obtenido el resultado final: ' + resultadoFinal
      }, falloCallback);
    }, falloCallback);
  }, falloCallback);
```

Con las funciones modernas, adjuntamos nuestras functiones callback a las promesas devueltas, formando una cadena de promesa:

```
hazAlgo().then(function (resultado) {
    return hazAlgoMas(resultado);
})
    .then(function (nuevoResultado) {
        return hazLaTerceraCosa(nuevoResultado);
    })
    .then(function (resultadoFinal) {
        console.log('Obtenido el resultado final: ' + resultadoFinal);
    })
    .catch(falloCallback);
```

Los argumentos a `then` son opcionales, y `catch(falloCallBack)` es un atajo para `then(null, falloCallBack)`. Es posible que veas esto expresado con funciones de flecha :

```
hazAlgo()
    .then(resultado => hazAlgoMas(resultado))
    .then(nuevoResultado => hazLaTerceraCosa(nuevoResultado))
    .then(resultadoFinal => {
        console.log(`Obtenido el resultado final: ${resultadoFinal}`);
    })
    .catch(falloCallback);
```

**Importante**: Devuelve siempre resultados, de otra forma las funciones callback no se encadenarán, y los errores no serán capturados.

### Encadenar después de una captura

Es posible encadenar después de un fallo - por ejemplo: un `catch`- lo que es útil para lograr nuevas acciones incluso después de una acción fallida en la cadena. Lea el siguiente ejemplo:

```
new Promise((resolver, rechazar) => {
    console.log('Inicial');

    resolver();
})
    .then(() => {
        throw new Error('Algo falló');

        console.log('Haz esto');
    })
    .catch(() => {
        console.log('Haz aquello');
    })
    .then(() => {
        console.log('Haz esto sin que importe lo que sucedió antes');
    });
```

Esto devolverá el siguiente texto:

```
Inicial
Haz aquello
Haz esto sin que importe lo que sucedió antes
```

Note que el texto "Haz esto" no es escrito porque el error "Algo falló" causó un rechazo.

## Propagación de errores

Tal vez recuerdes haber visto `falloCallback` tres veces en la pirámide en un ejemplo anterior, en comparación con sólo una vez al final de la cadena de promesas:

```
hazAlgo()
    .then(resultado => hazAlgoMas(valor))
    .then(nuevoResultado => hazLaTerceraCosa(nuevoResultado))
    .then(resultadoFinal => console.log(`Obtenido el resultado final:
${resultadoFinal}`))
    .catch(falloCallback);
```

Básicamente, una cadena de promesas se detiene si hay una excepción, y recorre la cadena buscando manejadores de captura. Lo siguiente está mucho más adaptado a la forma de trabajo del código síncrono:

```
try {
    let resultado = syncHazAlgo();
    let nuevoResultado = syncHazAlgoMas(resultado);
    let resultadoFinal = syncHazLaTerceraCosa(nuevoResultado);
    console.log(`Obtenido el resultado final: ${resultadoFinal}`);
} catch (error) {
    falloCallback(error);
}
```

Esta simetría con el código síncrono culmina con la mejora sintáctica async/await en ECMASCript 2017:

```
async function foo() {
    try {
        let resultado = await hazAlgo();
        let nuevoResultado = await hazAlgoMas(resultado);
        let resultadoFinal = await hazLaTerceraCosa(nuevoResultado);
        console.log(`Obtenido el resultado final: ${resultadoFinal}`);
    } catch (error) {
        falloCallback(error);
    }
}
```

Se construye sobre `promesas`, por ejemplo, `hazAlgo()` es la misma función que antes.

Las `promesas` resuelven un fallo fundamental de la pirámide de funciones callback, capturando todos los errores, incluso excepciones lanzadas y errores de programación. Esto es esencial para la composición funcional de operaciones asíncronas.

## Eventos de rechazo de Promesas

Cuando una `promesa` es rechazada, uno de los dos eventos se envía al ámbito global (generalmente, éste es el `window`, o, si se utiliza en un trabajador web, es el `Worker` u otra interfaz basada en un trabajador). Los dos eventos son:

**`rejectionhandled` (en-US)**

Se envía cuando se rechaza una promesa, una vez que el rechazo ha sido manejado por la función `reject` del ejecutor.

**`unhandledrejection` (en-US)**

Se envía cuando se rechaza una promesa pero no hay un controlador de rechazo disponible.

En ambos casos, el evento (del tipo `PromiseRejectionEvent` (en-US)) tiene como miembros una propiedad `promise` (en-US) que indica que la promesa fue rechazada, y una propiedad `reason` (en-US) que proporciona el motivo por el cuál se rechaza la promesa.

Esto hace posible ofrecer el manejo de errores de promesas, y también ayuda a depurarlos. Estos controladores son globales, por lo tanto, todos los errores serán manejados por éstos independientemente de la fuente.

**Un caso de especial utilidad**: al escribir código para Node.js, es común que los módulos que incluyas en tu proyecto no cuenten con un controlador de evento para promesas rechazadas. Estos se registran en la consola en tiempo de ejecución de Node. Puedes capturarlos para analizarlos y manejarlos en tu código - o solo evitar que abarroten tu salida - agregando un controlador para el evento `unhandledrejection` (en-US), como se muestra a continuación:

```
window.addEventListener("unhandledrejection", event => {
    /* Podrías comenzar agregando código para examinar
       la promesa específica analizando event.promise
       y la razón del rechazo, accediendo a event.reason */

    event.preventDefault();
}, false);
```

Llamando al método `preventDefault()` del evento, le dices a Javascript en tiempo de ejecución que no realice su acción predeterminada cuando las promesas rechazadas no cuenten con manejadores. En el caso de Node, esa acción predeterminada usualmente registra el error en la consola.

Lo ideal, por supuesto, sería examinar las promesas rechazadas para asegurarte que ninguna de ellas tiene errores de código reales antes de descartar esos eventos.

## Crear una promesa alrededor de una vieja API de callbacks

Una `Promise` puede ser creada desde cero usando su constructor. Esto debería ser sólo necesario para envolver viejas APIs.

En un mundo ideal, todas las funciones asíncronas devolverían promesas. Desafortunadamente, algunas APIs aún esperan que se les pase callbacks con

resultado fallido/exitoso a la forma antigua. El ejemplo más obvio es la función `setTimeout()`:

```
setTimeout(() => diAlgo("pasaron 10 segundos"), 10000);
```

Combinar callbacks del viejo estilo con promesas es problemático. Si `diAlgo` falla o contiene un error de programación, nada lo captura. La función `setTimeout` es culpable de esto.

Afortunadamente podemos envolverlas en una promesa. La mejor práctica es envolver las funciones problemáticas en el nivel más bajo posible, y después nunca llamarlas de nuevo directamente:

```
const espera = ms => new Promise(resuelve => setTimeout(resuelve, ms));

espera(10000).then(() => diAlgo("10 segundos")).catch(falloCallback);
```

Básicamente, el constructor de la promesa toma una función ejecutora que nos permite resolver o rechazar manualmente una promesa. Dado que `setTimeout` no falla realmente, descartamos el rechazo en este caso.

## Composición

`Promise.resolve()` y `Promise.reject()` son atajos para crear manualmente una promesa resuelta o rechazada respectivamente. Esto puede ser útil a veces.

`Promise.all()` son `Promise.race()` son dos herramientas de composición para ejecutar operaciones asíncronas en paralelo.

Podemos comenzar operaciones en paralelo y esperar que finalicen todas ellas de la siguiente manera:

```
Promise.all([func1(), func2(), func3()])
    .then(([resultado1, resultado2, resultado3]) => { /* usa resultado1,
resultado2 y resultado3 */ });
```

La composición secuencial es posible usando Javascript inteligente:

```
[func1, func2, func3].reduce((p, f) => p.then(f), Promise.resolve())
    .then(result3 => { /* use result3 */ });
```

Básicamente, reducimos un conjunto de funciones asíncronas a una cadena de promesas equivalente
a: `Promise.resolve().then(func1).then(func2).then(func3);`

Esto se puede convertir en una función de composición reutilizable, que es común en la programación funcional:

```
const aplicarAsync = (acc,val) => acc.then(val);
const componerAsync = (...funcs) => x => funcs.reduce(aplicarAsync,
Promise.resolve(x));
```

La función `componerAsync()` aceptará cualquier número de funciones como argumentos, y devolverá una nueva función que acepta un valor inicial que es pasado a través del conducto de composición. Esto es beneficioso porque cualquiera o todas las funciones pueden ser o asíncronas o síncronas y se garantiza que serán ejecutadas en el orden correcto:

```
const transformData = componerAsync(func1, asyncFunc1, asyncFunc2, func2);
const resultado3 = transformData(data);
```

En ECMAScript 2017, la composición secuencial puede ser realizada usando simplemente async/await:

```
let resultado;
for (const f of [func1, func2, func3]) {
    resultado = await f(resultado);
}
```

## Sincronización

Para evitar sorpresas, las funciones pasadas a `then()` nunca serán llamadas sincrónicamente, incluso con una promesa ya resuelta:

```
Promise.resolve().then(() => console.log(2));
console.log(1); // 1, 2
```

En lugar de ejecutarse inmediatamente, la función pasada es colocada en una cola de microtareas, lo que significa que se ejecuta más tarde cuando la cola es vaciada al final del actual ciclo de eventos de JavaScript:

```
const espera = ms => new Promise(resuelve => setTimeout(resuelve, ms));
```

```
espera().then(() => console.log(4));
Promise.resuelve().then(() => console.log(2)).then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

## Anidamiento

Las cadenas de promesas simples se mantienen planas sin anidar, ya que el anidamiento puede ser el resultado de una composición descuidada.

El anidamiento es una estructura de control para limitar el alcance de las sentencias `catch`. Específicamente, un `catch` anidado sólo captura fallos dentro de su contexto y por debajo, no captura errores que están más arriba en la cadena fuera del alcance del anidamiento. Cuando se usa correctamente, da mayor precisión en la recuperación de errores:

```
hacerAlgoCritico()
    .then(resultado => hacerAlgoOpcional()
        .then(resultadoOpcional => hacerAlgoSuper(resultadoOpcional))
        .catch(e => { })) // Ignorar si hacerAlgoOpcional falla.
    .then(() => masAsuntosCriticos())
    .catch(e => console.log("Acción crítica fallida: " + e.message));
```

Nota que aquí los pasos opcionales están anidados, por la precaria colocación de lo externo (y) alrededor de ellos.

La declaración interna `catch` solo detecta errores de `hacerAlgoOpcional()` y `hacerAlgoSuper()`, después de lo cuál el código se reanuda con `masAsuntosCriticos()`. Es importante destacar que si `hacerAlgoCritico()` falla, el error es capturado únicamente por el `catch` final.

## Errores comunes

Aquí hay algunos errores comunes que deben tenerse en cuenta al componer cadenas de promesas. Varios de estos errores se manifiestan en el siguiente ejemplo:

```
// ¡Mal ejemplo!
hacerlAlgo().then(function (resultado) {
    hacerOtraCosa(resultado) // Olvida devolver una promesa desde el interior
de la cadena + anidamiento innecesario
        .then(nuevoResultado => hacerUnaTerceraCosa(nuevoResultado));
}).then(() => hacerUnaCuartaCosa());
  // Olvida terminar la cadena con un catch!
```

El primer error es no encadenar las acciones adecuadamente. Esto sucede cuando creamos una promesa y olvidamos devolverla. Como consecuencia, la cadena se rompe, o mejor dicho, tenemos dos cadenas independientes que compiten. Esto significa que `hacerUnaCuartaCosa()` no esperará a que finalicen `hacerOtraCosa()` o `hacerUnaTerceraCosa()`, y se ejecutará paralelamente a ellas. Las cadenas separadas también tienen un manejador de errores separado, lo que provoca errores no detectados.

El segundo error es el anidamiento innecesario, que da lugar al primer error. La anidación también limita el alcance de los manejadores de errores internos, que - si no son deseados - pueden llevar a errores no detectados. Una variante de esto es el constructor anti-patrón de promesas, el cuál combina el anidamiento con el uso redundante del constructor de promesa para envolver el código que ya usa promesas.

El tercer error es olvidar cerrar las cadenas con catch. Las cadenas de promesas no terminadas conducen a errores no capturados en la mayoría de los navegadores.

Una buena regla es devolver o terminar siempre las cadenas de promesas, y tan pronto como obtenga una nueva promesa, devolverla de inmediato, para aplanar las cosas:

```
hacerAlgo()
    .then(function (resultado) {
        return hacerOtraCosa(resultado);
    })
    .then(nuevoResultado => hacerUnaTerceraCosa(nuevoResultado))
    .then(() => hacerUnaCuartaCosa())
    .catch(error => console.log(error));
```

Nota que `() => x` es un atajo para `() => { return x; }`.

Ahora tenemos una cadena determinística simple con un manejador de error adecuado.

El uso de async / await aborda la mayoría, si no todos estos problemas, la desventaja es que el error más común con esa sintaxis es olvidar la palabra clave await.

**Referencias**

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

https://www.freecodecamp.org/espanol/news/promesas-en-javascript-es6/

https://es.javascript.info/promise-basics

https://programacionymas.com/blog/promesas-javascript

https://javadesde0.com/promesas-promises-en-javascript-js/

https://rlbisbe.net/2015/04/26/explicando-promises-de-javascript-con-un-ejemplo-simple/