



# JavaScript

## Promesas y manejo de promesas

*< Ing := Carlos H. Rueda C++ >*

# 1. ¿Qué son las promesas?

Las promesas, en el contexto de la programación y la informática, son un concepto fundamental en el manejo de tareas asíncronas.

Son objetos que representan un valor que puede estar disponible ahora, en el futuro o nunca.



## ¿En que se usan?

Las promesas se utilizan para realizar operaciones asíncronas en lenguajes de programación, como JavaScript, y permiten manejar el flujo de control de manera más efectiva en situaciones en las que las operaciones pueden tomar un tiempo variable en completarse, como solicitudes de red, lectura de archivos o consultas a bases de datos.

## 00 Un caso de la vida real de cómo son las promesas



Imagina que estás esperando un paquete importante que se supone que debe entregarse en tu oficina. El **mensajero** de la compañía de envíos tenía que traerlo a una hora específica, pero cuando está a punto de llegar, se da cuenta de que olvidó el paquete en su oficina.

Sería una situación frustrante, ¿verdad?

\* \* \* \* \*





Sin embargo, el mensajero no se rinde.  
Afortunadamente, tiene un compañero que todavía  
está en la oficina en ese momento.

El mensajero llama a su compañero con urgencia y le  
pide ayuda, prácticamente le ruega que le ayude a  
encontrar el paquete. Su compañero **PROMETE** enviar  
un mensaje tan pronto tenga una actualización.





Si el paquete es encontrado finalmente, podría enviar un mensaje diciendo:

***"¡Listo, encontré tu paquete y te lo voy a enviar!"***

Pero si no puede encontrarlo, enviará un mensaje explicando la razón:

***"Lo siento, no pude encontrar enviar tu paquete porque se confundieron con la dirección."***





Mientras tanto, tú continúas trabajando en tu oficina mientras esperas el paquete. Estás aferrado a la promesa de que el paquete será entregado.

El mensajero responde a tus preguntas y solicitudes telefónicas, pero en última instancia, tu satisfacción depende del **ESTADO FINAL** del paquete.

\* \* \* \* \*





Finalmente, su compañero le envía un mensaje. Como habían acordado previamente, si no encuentra el paquete, te lo dirá junto con la razón de la pérdida.

Si eso sucede, podrías estar decepcionado y frustrado. Por otro lado, si su compañero encuentra el paquete, estarás aliviado y contento de que cumplió con su compromiso y puedes seguir adelante con tus planes que dependían del contenido del paquete.

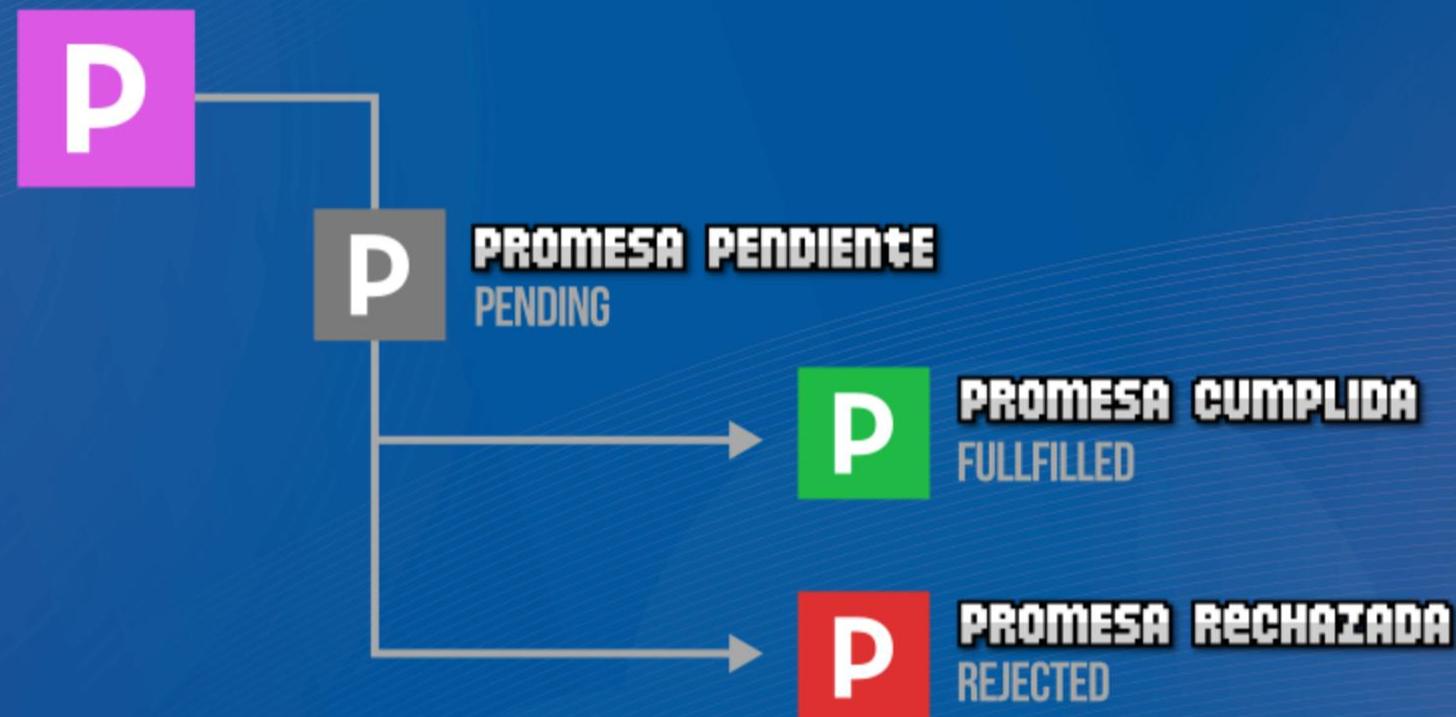


## 1.2 Estados de las promesa

Las promesas tiene tres estado principales:

1. **Pendiente (pending)**: Inicialmente, una promesa se encuentra en estado pendiente, lo que significa que la operación aún no se ha completado.
2. **Resuelta (fulfilled)**: Cuando la operación asíncrona se completa con éxito, la promesa pasa al estado resuelto y proporciona un valor resultante.
3. **Rechazada (rejected)**: Si la operación asíncrona falla, la promesa pasa al estado rechazado y proporciona un motivo o razón para el fallo.

# PROMESAS



\* \* \* \* \*

El uso de promesas facilita la escritura de código asíncrono más legible y mantenible, ya que permite encadenar múltiples operaciones asíncronas y manejar errores de manera más eficaz. Además, las promesas son la base de la programación asíncrona en muchos lenguajes de programación modernos y son ampliamente utilizadas en desarrollo web y aplicaciones que requieren interacciones asíncronas con servidores y recursos externos.

\* \* \* \* \*

## 00 Ejemplo del paquete y el mensajero con los estados de las promesas:

Este ejemplo se relaciona con una promesa en JavaScript de la siguiente manera:

1. El paquete importante que estás esperando representa una tarea **asincrónica** o una operación que debe **completarse en el futuro**.
2. El mensajero que olvidó el paquete es como una **promesa que aún no se ha resuelto**. La promesa representa la tarea pendiente de entregarte el resultado.



3. El llamado del mensajero a su compañero para buscar el paquete es como \*\*una promesa que se inicia\*\* con la expectativa de que se resolverá en algún momento. El envío del mensaje prometiendo una actualización es equivalente a la \*\*creación de una promesa\*\* en JavaScript.



\* \* \* \* \*



4. El estado final del paquete, ya sea encontrado o perdido, se asemeja al **“estado final de una promesa”** en JavaScript, que puede ser resuelta exitosamente (con un valor) o rechazada (con un motivo).

5. Tu satisfacción depende del resultado de la promesa, al igual que en JavaScript, el **manejo de la promesa depende de si se resuelve con éxito o no.**



\* \* \* \* \*



6. La promesa de JavaScript utiliza las funciones **fulfilled** y **rejected** para indicar si se completó la tarea asíncrona con éxito o si se produjo un error, de manera similar a cómo el compañero del mensajero envía un mensaje indicando si encontró el paquete o la razón de la pérdida.



\* \* \* \* \*



## 1.3 Promesas en Javascript

Las promesas en JavaScript son objetos que representan valores que pueden estar disponibles ahora, en el futuro o nunca.

Se utilizan para manejar tareas asíncronas, como solicitudes de red, lectura de archivos o consultas a bases de datos.



# Estados de las promesas en JS

Las promesas pueden estar en uno de tres estados: pendiente (**pending**), aceptada (**fulfilled**) o rechazada (**rejected**).

Los principales métodos para trabajar con promesas son:

Método	Descripción
.then(resolve)	Maneja la ejecución exitosa de una promesa, se ejecuta cuando la promesa se resuelve con éxito.
.catch(reject)	Maneja el rechazo de una promesa, se ejecuta cuando la promesa es rechazada.

## Metodos de las promesas (*Continuacion*)

Método	Descripción
.then(resolve, reject)	Combina el manejo de casos exitosos y de error en un solo .then().
.finally(end)	Permite ejecutar una función <b>end</b> tanto si la promesa se cumple como si se rechaza. Se utiliza para acciones que deben ocurrir sin importar el resultado de la promesa.

\* \* \* \* \*



## 2. Consumir una promesa

La forma general de consumir una promesa es utilizando el método `.then()` con un solo parámetro, ya que en muchas ocasiones, lo que nos interesa es ejecutar una acción cuando la promesa se cumple.

Esto simplifica la sintaxis y permite centrarse en la lógica que se ejecutará al completarse la promesa sin preocuparse por el manejo de errores en ese momento.



Por ejemplo:

```
miPromesa
  .then(resultado => {
    // Realizar una acción cuando la promesa se cumple
    console.log("La promesa se ha cumplido con éxito:", resultado);
  });

```

En este caso, el código dentro del bloque `**.then()` se ejecutará solo cuando la promesa se cumpla exitosamente, lo que simplifica el manejo de casos exitosos.

\* \* \* \* \*



### 3. Código no bloqueante

Algo de gran importancia, que a veces pasamos por alto, es que el código que se encuentra dentro de un bloque `**.then()` es asincrónico y no bloqueante:

**Asincrónico:** Esto significa que es probable que no se ejecute de inmediato, sino que puede tomar un tiempo para su ejecución.

**No bloqueante:** Mientras espera su ejecución, no impide que el resto del programa continúe su flujo de trabajo.

## ¿Qué implica esto?

Que cuando se llega a un bloque `**.then()`, el sistema no se bloquea; en su lugar, coloca la función en "espera" hasta que la promesa se cumpla. Mientras tanto, el programa sigue procesando las tareas restantes.

```
// Crear una promesa que simula una operación asíncrona
const miPromesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    // Resuelve la promesa después de un retraso simulado
    resolve("¡Operación completada con éxito!");
  }, 2000);
});

console.log("Inicio de la operación");

miPromesa
  .then(resultado => {
    console.log(resultado); // Se ejecuta cuando la promesa se cumple
  })
  .catch(error => {
    console.error("Error:", error); // Manejar errores si la promesa se rechaza
  });

console.log("Tareas adicionales"); // Se ejecuta antes de que la promesa se complete
```

En este ejemplo, la promesa `**miPromesa**` simula una operación asíncrona que se resuelve después de un retraso de 2 segundos. Mientras se espera la resolución de la promesa, las líneas de código fuera del `**.then()` no se bloquean y siguen ejecutándose, como se muestra en la salida.

Esto demuestra el concepto de código no bloqueante en la programación asíncrona.



i DevTools is now available in Spanish! Always match Chrome's language [Switch DevTools to Spanish](#) [Don't show again](#)

Elements Console Sources Network Performance Memory Application Security Lighthouse

one-google-bar Filter

```
> const miPromesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("¡Operación completada con éxito!"); // Resuelve la promesa después de un retraso simulado
  }, 2000);
});

console.log("Inicio de la operación");

miPromesa
  .then(resultado => {
    console.log(resultado); // Se ejecuta cuando la promesa se cumple
  })
  .catch(error => {
    console.error("Error:", error); // Manejar errores si la promesa se rechaza
  });

console.log("Tareas adicionales"); // Se ejecuta antes de que la promesa se complete

// Mientras se espera la resolución de la promesa, el programa continúa ejecutando "Tareas adicionales",
// lo que demuestra que el código no bloquea la ejecución.

Inicio de la operación
Tareas adicionales
< undefined
¡Operación completada con éxito!
> |
```

# 4. Crear promesas

Puedes crear una promesa en JavaScript utilizando el constructor **\*\*Promise\*\***. Una promesa se crea alrededor de una función que toma dos argumentos: **resolve** y **reject**. La función se ejecuta de forma asíncrona y, en función de las circunstancias, se llama a **resolve** cuando la promesa se cumple con éxito o a **reject** cuando la promesa se rechaza.

# Aquí tienes un ejemplo de cómo crear una promesa:

```
// Crear una promesa
const miPromesa = new Promise((resolve, reject) => {
  // Simulamos una operación asíncrona que toma tiempo en completarse
  setTimeout(() => {
    const exito = true; // Cambiar a false para simular un rechazo
    if (exito) {
      resolve("¡La promesa se cumplió con éxito!");
    } else {
      reject("Hubo un error al cumplir la promesa <()");
    }
  }, 2000); // Simulamos un retraso de 2 segundos
});

// Consumir la promesa
miPromesa
  .then(resultado => {
    console.log("Éxito:", resultado); // Manejar el caso de éxito
  })
  .catch(error => {
    console.error("Error:", error); // Manejar el caso de error
 });
```

1. Creamos una promesa llamada `miPromesa` que simula una operación asincrónica utilizando `setTimeout`. La promesa se resuelve después de un retraso de 2 segundos.
2. Dentro de la función de la promesa, utilizamos una variable `exito` para simular si la operación se cumplirá con éxito (cambia a `false` para simular un rechazo).

3. Si `exito` es `true`, llamamos a `resolve` con un mensaje de éxito. De lo contrario, llamamos a `reject` con un mensaje de error.
4. Luego, consumimos la promesa utilizando los métodos `.then()` y `.catch()`. El método `.then()` maneja el caso de éxito y muestra el mensaje "Éxito". El método `.catch()` maneja el caso de error y muestra el mensaje de error en la consola.

\* \* \* \* \*



Importar marcadores... 🔥 Comenzar a usar Firefox

Cursor Inspector Consola Depurador Red Editor de estilo R

Trash Salida del filtro Errores Advertencias Registros Información D

```
» ▶ // Crear una promesa
const miPromesa = new Promise((resolve, reject) => {
    // Simulamos una operación asíncrona que toma tiempo en completarse
    setTimeout(() => {
        const exito = true; // Cambiar a false para simular un rechazo...
    
```

```
← ▶ Promise { <state>: "pending" }
```

Éxito: ¡La promesa se cumplió con éxito!

```
»
```