### [**https://shorturl.at/vrRp6**](https://shorturl.at/vrRp6)

### **Understanding Promises in JavaScript**

In JavaScript, a **Promise** is an object that represents the eventual outcome of an asynchronous operation. It can exist in one of three states:

1. **Pending**: The initial state; the operation has not yet completed.
2. **Fulfilled**: The operation completed successfully, and a result is available.
3. **Rejected**: The operation failed, and an error is available.

#### **Using resolve and reject**

* **resolve(value)**: Call this method to indicate that the asynchronous operation completed successfully. You can pass any data type (like objects, arrays, strings, etc.) that you want to return to the caller.  
  + Example: If you fetch user data successfully, you might call resolve(userData) to return the user information.
* **reject(error)**: Call this method when the operation fails for any reason. You can pass an error message or an Error object, which the caller can handle using .catch().  
  + Example: If an operation fails because of a network issue, you might call reject("Network error").

#### **Benefits of Using Promises**

Promises provide a cleaner alternative for handling asynchronous operations compared to traditional callback functions. They allow developers to manage asynchronous code more effectively, leading to easier-to-read and maintainable code.

### **Key Features of Promises**

* **Flexible Resolved Values**: The values returned by resolved promises can be of any data type (strings, objects, arrays, or even other promises).
* **Chaining with .then()**: The .then() method is called on a promise to handle successful resolution, enabling developers to work with the returned value.

Example:  
 fetchData() .then(data => console.log(data));

* **Error Handling with .catch()**: The .catch() method is used to handle any errors that occur during the promise's execution. This provides a robust way to manage exceptions.

Example:  
 fetchData().catch(error => console.error(error));

### **Example 1: Resolving with a String**

**Description**: This example demonstrates a simple promise that simulates returning a greeting string after a delay. The promise resolves successfully with a string value.

function sayHello() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const greeting = "Hello, World!";

resolve(greeting); // Resolving with a string

}, 1000); // Simulates a 1-second delay

});

}

sayHello()

.then(message => {

console.log(message); // Output: Hello, World!

})

.catch(error => {

console.log(error); // This runs if the Promise is rejected

});

**then() Method**:

* The .then() method is called on a Promise and is used to handle the resolved value of that Promise.
* It takes one or two arguments:
  + The first argument is a callback function that receives the resolved value, which you can use to perform further actions.
  + The second argument (optional) is a callback function that receives any errors if the Promise is rejected (though it’s more common to handle errors using .catch()).
  + **Example**:
  + sayHello()
  + .then(result => {
  + console.log(result); // This runs if the Promise is resolved
  + })
  + .catch(error => {
  + console.log(error); // This runs if the Promise is rejected
  + });

### **Example 2: Resolving with an Object**

**Description**: This example simulates fetching a user object. After a delay, the promise resolves with an object containing user information.

function getUser() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const user = { id: 1, name: "Alice" }; // Object to resolve

resolve(user);

}, 1000); // Simulates a 1-second delay

});

}

getUser()

.then(user => {

console.log(user); // Output: { id: 1, name: "Alice" }

});

**Example 3: Rejecting with an Error Object**

**Description**: This example simulates a data fetching operation that might fail. If the success condition is false, the promise is rejected with an Error object.

function fetchData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const success = false; // Change this to true to simulate success

if (success) {

resolve("Data fetched successfully!");

} else {

reject(new Error("Failed to fetch data")); // Rejecting with an Error object

}

}, 1000); // Simulates a 1-second delay

});

}

fetchData()

.catch(error => {

console.error(error.message); // Output: Failed to fetch data

});

### **Example 4: Resolving with an Array**

**Description**: This example simulates fetching a list of items. The promise resolves successfully with an array of fruit names.

function fetchItems() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const items = ["Apple", "Banana", "Cherry"]; // Resolving with an array

resolve(items);

}, 1000); // Simulates a 1-second delay

});

}

fetchItems()

.then(items => {

console.log(items); // Output: [ 'Apple', 'Banana', 'Cherry' ]

});

### **Example 5: Rejecting with a String**

**Description**: This example demonstrates a division operation that may fail if the denominator is zero. The promise is rejected with a string message if division by zero is attempted.

function divideNumbers(num1, num2) {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (num2 === 0) {

// Rejecting with a string message

reject("Cannot divide by zero");

} else {

const result = num1 / num2;

resolve(result); // Resolving with the division result

}

}, 1000); // Simulates a 1-second delay

});

}

divideNumbers(10, 0)

.catch(error => {

console.error(error); // Output: Cannot divide by zero

});

### **Example 6: Resolving with Another Promise**

**Description**: This example simulates fetching profile data and user data in parallel using Promise.all(). The promise resolves with an object containing both results after both promises are fulfilled.

function fetchProfile() {

return new Promise((resolve) => {

setTimeout(() => {

resolve("Profile data"); // Resolving with a string

}, 1000); // Simulates a 1-second delay

});

}

function fetchUser() {

return new Promise((resolve) => {

setTimeout(() => {

resolve({ id: 1, name: "Bob" }); // Resolving with an object

}, 1000); // Simulates a 1-second delay

});

}

function fetchCombinedData() {

return Promise.all([fetchProfile(), fetchUser()])

.then(([profile, user]) => {

// Callback: Resolving with an object containing both results

return { profile, user };

});

}

fetchCombinedData()

.then(data => {

console.log(data);

});

// Output: { profile: 'Profile data', user: { id: 1, name: 'Bob' } }

// Use an async function to assign the value to x

async function main() {

let x = await fetchCombinedData(); // Wait for the promise to resolve

console.log(x); // Now x contains the resolved data

}

main(); // Call the async function