**JavaScript CallBack Function**

In this tutorial, you will learn about JavaScript callback functions with the help of examples.

A function is a block of code that performs a certain task when called. For example,

// function

function greet(name) {

console.log('Hi' + ' ' + name);

}

greet('Peter'); // Hi Peter

[Run Code](https://www.programiz.com/javascript/online-compiler)

In the above program, a string value is passed as an argument to the greet() function.

In JavaScript, you can also pass a function as an argument to a function. This function that is passed as an argument inside of another function is called a callback function. For example,

// function

function greet(name, callback) {

console.log('Hi' + ' ' + name);

callback();

}

// callback function

function callMe() {

console.log('I am callback function');

}

// passing function as an argument

greet('Peter', callMe);

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Output**

Hi Peter

I am callback function

In the above program, there are two functions. While calling the greet() function, two arguments (a string value and a function) are passed.

The callMe() function is a callback function.

**Benefit of Callback Function**

The benefit of using a callback function is that you can wait for the result of a previous function call and then execute another function call.

In this example, we are going to use the setTimeout() method to mimic the program that takes time to execute, such as data coming from the server.

**Example: Program with setTimeout()**

// program that shows the delay in execution

function greet() {

console.log('Hello world');

}

function sayName(name) {

console.log('Hello' + ' ' + name);

}

// calling the function

setTimeout(greet, 2000);

sayName('John');

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Output**

Hello John

Hello world

As you know, the [setTimeout()](https://www.programiz.com/javascript/setTimeout) method executes a block of code after the specified time.

Here, the greet() function is called after **2000** milliseconds (**2** seconds). During this wait, the sayName('John'); is executed. That is why Hello John is printed before Hello world.

The above code is executed asynchronously (the second function; sayName() does not wait for the first function; greet() to complete).

**Example: Using a Callback Function**

In the above example, the second function does not wait for the first function to be complete. However, if you want to wait for the result of the previous function call before the next statement is executed, you can use a callback function. For example,

// Callback Function Example

function greet(name, myFunction) {

console.log('Hello world');

// callback function

// executed only after the greet() is executed

myFunction(name);

}

// callback function

function sayName(name) {

console.log('Hello' + ' ' + name);

}

// calling the function after 2 seconds

setTimeout(greet, 2000, 'John', sayName);

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Output**

Hello world

Hello John

In the above program, the code is executed synchronously. The sayName() function is passed as an argument to the greet() function.

The setTimeout() method executes the greet() function only after **2** seconds. However, the sayName() function waits for the execution of the greet() function.

**Note**: The callback function is helpful when you have to wait for a result that takes time. For example, the data coming from a server because it takes time for data to arrive.

A callback is a function passed as an argument to another function.

Using a callback, you could call the calculator function (myCalculator) with a callback (myCallback), and let the calculator function run the callback after the calculation is finished:

Example

function myDisplayer(some) {  
  document.getElementById("demo").innerHTML = some;  
}  
  
function myCalculator(num1, num2, myCallback) {  
  let sum = num1 + num2;  
  myCallback(sum);  
}  
  
myCalculator(5, 5, myDisplayer);

In the example above, myDisplayer is a called a **callback function**.

It is passed to myCalculator() as an **argument**.

Note

When you pass a function as an argument, remember not to use parenthesis.

Right: myCalculator(5, 5, myDisplayer);

Wrong: ;

Example

// Create an Array  
const myNumbers = [4, 1, -20, -7, 5, 9, -6];  
  
// Call removeNeg with a callback  
const posNumbers = removeNeg(myNumbers, (x) => x >= 0);  
  
// Display Result  
document.getElementById("demo").innerHTML = posNumbers;  
  
// Keep only positive numbers  
function removeNeg(numbers, callback) {  
  const myArray = [];  
  for (const x of numbers) {  
    if (callback(x)) {  
      myArray.push(x);  
    }  
  }  
  return myArray;  
}

**What is Synchronous in JavaScript?**

As its base JavaScript language is synchronous. Synchronous means the code runs in a particular sequence of instructions given in the program. Each instruction waits for the previous instruction to complete its execution.

See the following code example of synchronous JavaScript - **Code Example -**

console.log('First');

console.log('Second');

console.log('Third');

**Output -**

First

Second

Third

In the above JavaScript code snippet, three lines of instructions are given. Every instruction runs once after the previous instruction gets executed. Due to this synchronous nature of javascript, we get the output of console logs in the sequence we provided in the program.

When we say synchronous vs asynchronous JavaScript the execution sequence of instructions in a program is what differentiates them.

**What is Asynchronous in JavaScript?**

As we saw in the synchronous code example, all instructions in the program execute one after another, and every instruction waits for its previous instruction to get executed. Due to this nature of synchronous programming, sometimes important instructions get blocked due to some previous instructions, which causes a delay in the user interface. Asynchronous code execution allows to execution next instructions immediately and doesn't block the flow because of previous instructions.

See the following coding example to understand how javascript works asynchronously -

**Code Example -**

console.log('First');

*// Set timeout for 2 seconds*

setTimeout(() => console.log('Second'), 2000);

console.log('Third');

**Output -**

First

Third

Second

As we can see in the output of the above code example, Third gets printed before Second, because of the asynchronous execution of the code. Here setTimeout() function waits for 2 seconds, and in the meantime, the next instruction gets executed without waiting for the previous one to complete the execution.

**What is the Difference Between Synchronous vs Asynchronous JavaScript?**

| **Sr. no.** | **Synchronous JavaScript** | **Asynchronous JavaScript** |
| --- | --- | --- |
| 1 | Instruction in synchronous code executes in a given sequence. | Instructions in asynchronous code can execute in parallel. |
| 2 | Each operation waits for the previous operation to complete its execution. | Next operation can occur while the previous operation is still getting processed. |
| 3 | Most of the time JavaScript is used as Synchronous code. | Asynchronous JavaScript is preferred in situations in which execution gets blocked indefinitely. |

Callbacl example1:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<script>

function show1(a)

{

document.write("<br>function 1 is required when i call function 2 :- " + a);

}

function show2(a,b)

{

show1(a);

document.write("<br>function :- "+ b);

}

show2(2,3);

show2(4,5);

</script>

</body>

</html>

Example2:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<p id="demo1">somthing</p>

<script>

setTimeout(fun1 ,6000);

function fun1()

{

// document.getElementById("demo1").innerHTML="Somthing..";

document.getElementById("demo1").style.background="red";

}

</script>

</body>

</html>

Example3:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<!-- <button onclick="change()"> Click</button> -->

<p id="d1"></p>

<script>

// setTimeout(function(){

// // document.getElementById("demo1").innerHTML="Somthing..";

// document.getElementById("demo1").style.background="red";

// } ,6000);

function change()

{

setTimeout(() => {

var d1=new Date();

document.getElementById("d1").innerHTML=d1.getHours()+":"+d1.getMinutes()+":"+d1.getSeconds();

}, 1000);

}

change();

</script>

</body>

</html>

Example4:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

</head>

<body>

<h2 id="d1"></h2>

<script>

setInterval(()=>{

var d1=new Date();

document.getElementById("d1").innerHTML=d1.getHours()+":"+d1.getMinutes()+":"+d1.getSeconds();

},1000);

</script>

</body>

</html>

**JavaScript Promise and Promise Chaining**

In this tutorial, you will learn about JavaScript promises and promise chaining with the help of examples.

In JavaScript, a promise is a good way to handle **asynchronous** operations. It is used to find out if the asynchronous operation is successfully completed or not.

A promise may have one of three states.

* Pending
* Fulfilled
* Rejected

A promise starts in a pending state. That means the process is not complete. If the operation is successful, the process ends in a fulfilled state. And, if an error occurs, the process ends in a rejected state.

For example, when you request data from the server by using a promise, it will be in a pending state. When the data arrives successfully, it will be in a fulfilled state. If an error occurs, then it will be in a rejected state.

**Create a Promise**

To create a promise object, we use the Promise() constructor.

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

//do something

});

The Promise() constructor takes a function as an argument. The function also accepts two functions resolve() and reject().

If the promise returns successfully, the resolve() function is called. And, if an error occurs, the reject() function is called.

Let's suppose that the program below is an asynchronous program. Then the program can be handled by using a promise.

**Example 1: Program with a Promise**

const count = true;

let countValue = new Promise(function (resolve, reject) {

if (count) {

resolve("There is a count value.");

} else {

reject("There is no count value");

}

});

console.log(countValue);

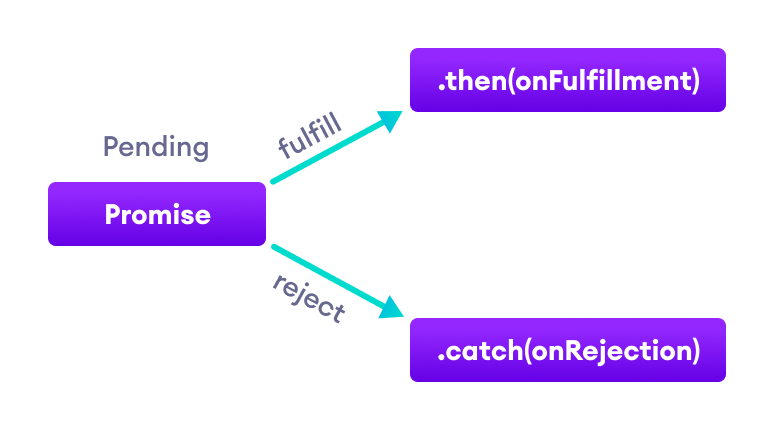
[Run Code](https://www.programiz.com/javascript/online-compiler)

**Output**

Promise {<resolved>: "There is a count value."}

In the above program, a Promise object is created that takes two functions: resolve() and reject(). resolve() is used if the process is successful and reject() is used when an error occurs in the promise.

The promise is resolved if the value of count is true.

Working of JavaScript promise

**JavaScript Promise Chaining**

Promises are useful when you have to handle more than one asynchronous task, one after another. For that, we use promise chaining.

You can perform an operation after a promise is resolved using methods then(), catch() and finally().

**JavaScript then() method**

The then() method is used with the callback when the promise is successfully fulfilled or resolved.

The syntax of then() method is:

promiseObject.then(onFulfilled, onRejected);

**Example 2: Chaining the Promise with then()**

// returns a promise

let countValue = new Promise(function (resolve, reject) {

resolve("Promise resolved");

});

// executes when promise is resolved successfully

countValue

.then(function successValue(result) {

console.log(result);

})

.then(function successValue1() {

console.log("You can call multiple functions this way.");

});

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Output**

Promise resolved

You can call multiple functions this way.

In the above program, the then() method is used to chain the functions to the promise. The then() method is called when the promise is resolved successfully.

You can chain multiple then() methods with the promise.

**JavaScript catch() method**

The catch() method is used with the callback when the promise is rejected or if an error occurs. For example,

// returns a promise

let countValue = new Promise(function (resolve, reject) {

reject('Promise rejected');

});

// executes when promise is resolved successfully

countValue.then(

function successValue(result) {

console.log(result);

},

)

// executes if there is an error

.catch(

function errorValue(result) {

console.log(result);

}

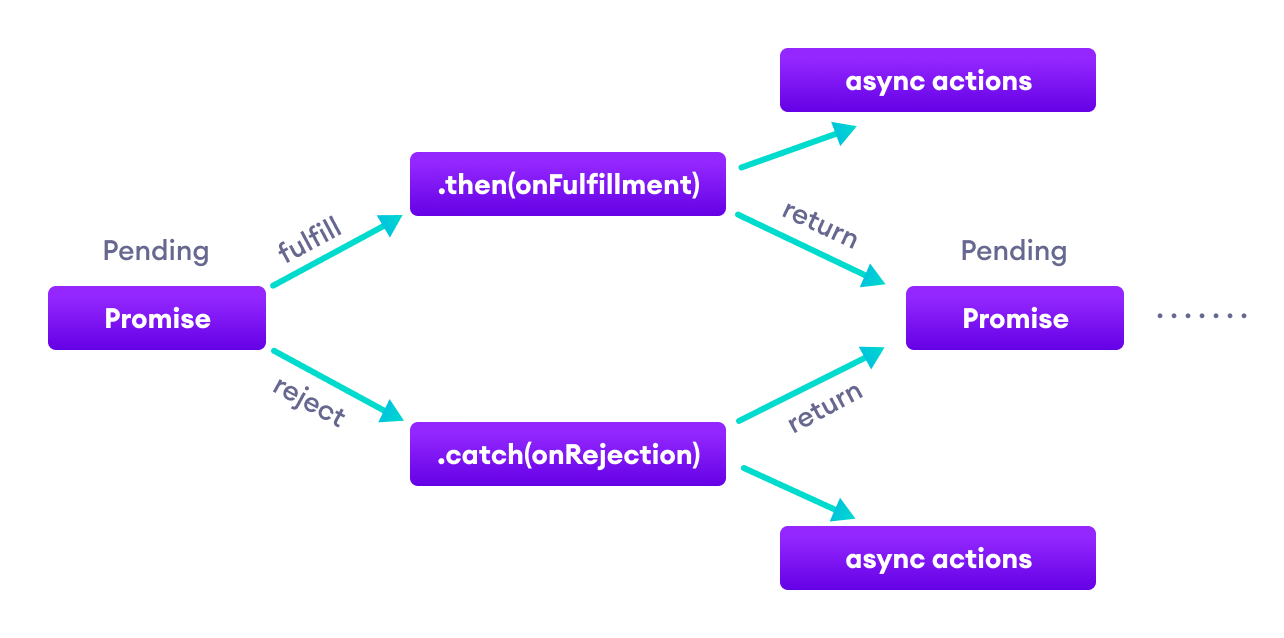
);

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Output**

Promise rejected

In the above program, the promise is rejected. And the catch() method is used with a promise to handle the error.

Working of JavaScript promise chaining

**JavaScript Promise Versus Callback**

Promises are similar to [callback functions](https://www.programiz.com/javascript/callback) in a sense that they both can be used to handle asynchronous tasks.

JavaScript callback functions can also be used to perform synchronous tasks.

Their differences can be summarized in the following points:

**JavaScript Promise**

1. The syntax is user-friendly and easy to read.
2. Error handling is easier to manage.
3. **Example:**
4. api().then(function(result) {
5. return api2() ;
6. }).then(function(result2) {
7. return api3();
8. }).then(function(result3) {
9. // do work
10. }).catch(function(error) {
11. //handle any error that may occur before this point

});

**JavaScript Callback**

1. The syntax is difficult to understand.
2. Error handling may be hard to manage.
3. **Example:**
4. api(function(result){
5. api2(function(result2){
6. api3(function(result3){
7. // do work
8. if(error) {
9. // do something
10. }
11. else {
12. // do something
13. }
14. });
15. });

});

**JavaScript finally() method**

You can also use the finally() method with promises. The finally() method gets executed when the promise is either resolved successfully or rejected. For example,

// returns a promise

let countValue = new Promise(function (resolve, reject) {

// could be resolved or rejected

resolve('Promise resolved');

});

// add other blocks of code

countValue.finally(

function greet() {

console.log('This code is executed.');

}

);

[Run Code](https://www.programiz.com/javascript/online-compiler)

**Output**

This code is executed.

**JavaScript Promise Methods**

There are various methods available to the Promise object.

|  |  |
| --- | --- |
| Method | Description |
| all(iterable) | Waits for all promises to be resolved or any one to be rejected |
| allSettled(iterable) | Waits until all promises are either resolved or rejected |
| any(iterable) | Returns the promise value as soon as any one of the promises is fulfilled |
| race(iterable) | Wait until any of the promises is resolved or rejected |
| reject(reason) | Returns a new Promise object that is rejected for the given reason |
| resolve(value) | Returns a new Promise object that is resolved with the given value |
| catch() | Appends the rejection handler callback |
| then() | Appends the resolved handler callback |
| finally() | Appends a handler to the promise |

**Javscript async/await**

In this tutorial, you will learn about JavaScript async/await keywords with the help of examples.

We use the async keyword with a function to represent that the function is an asynchronous function. The async function returns a [promise](https://www.programiz.com/javascript/promise).

The syntax of async function is:

async function name(parameter1, parameter2, ...paramaterN) {

// statements

}

Here,

* **name** - name of the function
* **parameters** - parameters that are passed to the function

**Example: Async Function**

// async function example

async function f() {

console.log('Async function.');

return Promise.resolve(1);

}

f();

**Output**

Async function.

In the above program, the async keyword is used before the function to represent that the function is asynchronous.

Since this function returns a promise, you can use the chaining method then() like this:

async function f() {

console.log('Async function.');

return Promise.resolve(1);

}

f().then(function(result) {

console.log(result)

});

**Output**

Async function

1

In the above program, the f() function is resolved and the then() method gets executed.

**JavaScript await Keyword**

The await keyword is used inside the async function to wait for the asynchronous operation.

The syntax to use await is:

let result = await promise;

The use of await pauses the async function until the promise returns a result (resolve or reject) value. For example,

// a promise

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

setTimeout(function () {

resolve('Promise resolved')}, 4000);

});

// async function

async function asyncFunc() {

// wait until the promise resolves

let result = await promise;

console.log(result);

console.log('hello');

}

// calling the async function

asyncFunc();

**Output**

Promise resolved

hello

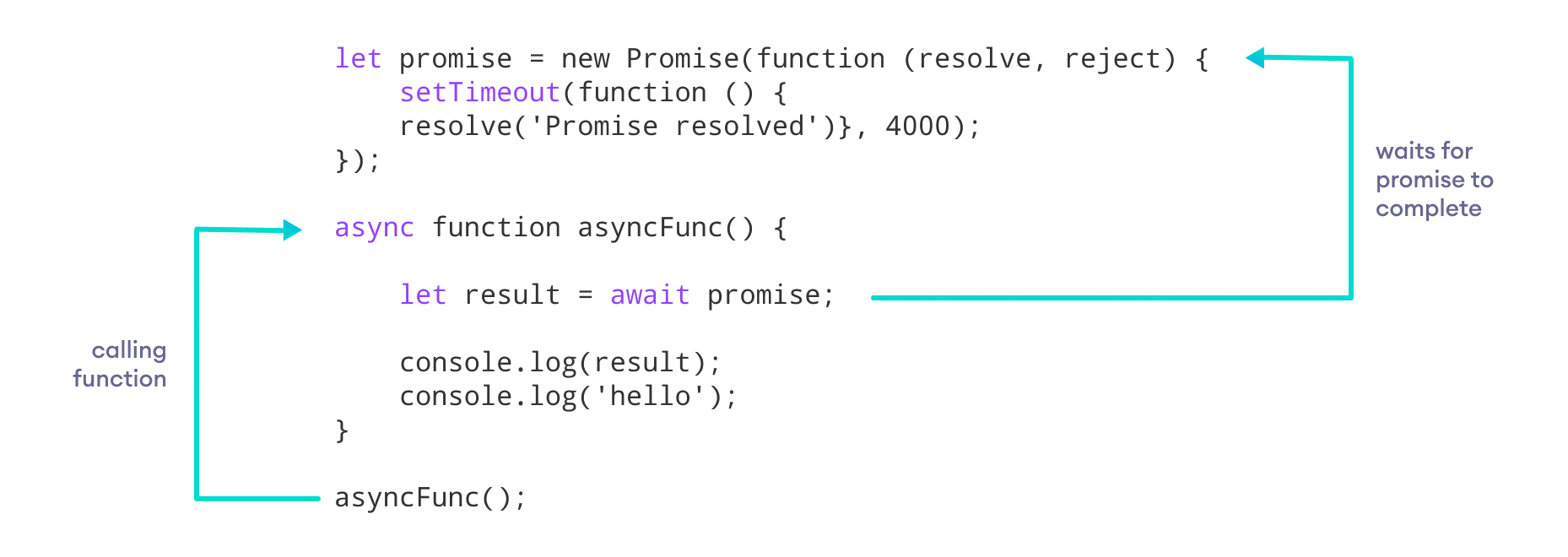
In the above program, a Promise object is created and it gets resolved after **4000** milliseconds. Here, the asyncFunc() function is written using the async function.

The await keyword waits for the promise to be complete (resolve or reject).

let result = await promise;

Hence, hello is displayed only after promise value is available to the result variable.

In the above program, if await is not used, hello is displayed before Promise resolved.

Working of async/await function

**Note**: You can use await only inside of async functions.

The async function allows the asynchronous method to be executed in a seemingly synchronous way. Though the operation is asynchronous, it seems that the operation is executed in synchronous manner.

This can be useful if there are multiple promises in the program. For example,

let promise1;

let promise2;

let promise3;

async function asyncFunc() {

let result1 = await promise1;

let result2 = await promise2;

let result3 = await promise3;

console.log(result1);

console.log(result1);

console.log(result1);

}

In the above program, await waits for each promise to be complete.

**Error Handling**

While using the async function, you write the code in a synchronous manner. And you can also use the catch() method to catch the error. For example,

asyncFunc().catch(

// catch error and do something

)

The other way you can handle an error is by using try/catch block. For example,

// a promise

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

setTimeout(function () {

resolve('Promise resolved')}, 4000);

});

// async function

async function asyncFunc() {

try {

// wait until the promise resolves

let result = await promise;

console.log(result);

}

catch(error) {

console.log(error);

}

}

// calling the async function

asyncFunc(); // Promise resolved

In the above program, we have used try/catch block to handle the errors. If the program runs successfully, it will go to the try block. And if the program throws an error, it will go to the catch block.

To learn more about try/catch in detail, visit JavaScript [JavaScript try/catch](https://www.programiz.com/javascript/try-catch-finally).

**Benefits of Using async Function**

* The code is more readable than using a [callback](https://www.programiz.com/javascript/callback) or a [promise](https://www.programiz.com/javascript/promise).
* Error handling is simpler.
* Debugging is easier.

**Note**: These two keywords async/await were introduced in the newer version of JavaScript (ES8). Some older browsers may not support the use of async/await.