Async and Await

**async function**

The **async function** declaration creates a [binding](https://developer.mozilla.org/en-US/docs/Glossary/Binding) of a new async function to a given name. The await keyword is permitted within the function body, enabling asynchronous, promise-based behavior to be written in a cleaner style and avoiding the need to explicitly configure promise chains.

function resolveAfter2Seconds() {

return new Promise((resolve) => {

setTimeout(() => {

resolve('resolved');

}, 2000);

});

}

async function asyncCall() {

console.log('calling');

const result = await resolveAfter2Seconds();

console.log(result);

// Expected output: "resolved"

}

asyncCall();

Syntax

JS

async function name(param0) {

statements

}

async function name(param0, param1) {

statements

}

async function name(param0, param1, /\* …, \*/ paramN) {

statements

}

Async Arrow Function

const AsyncFun = async () =>{

statements

}

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#parameters)

name

The function's name.

param Optional

The name of a formal parameter for the function. For the parameters' syntax, see the [Functions reference](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#function_parameters).

statements Optional

The statements comprising the body of the function. The await mechanism may be used.

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#description)

An async function declaration creates an [AsyncFunction](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AsyncFunction) object. Each time when an async function is called, it returns a new [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) which will be resolved with the value returned by the async function, or rejected with an exception uncaught within the async function.

Async functions can contain zero or more [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) expressions. Await expressions make promise-returning functions behave as though they're synchronous by suspending execution until the returned promise is fulfilled or rejected. The resolved value of the promise is treated as the return value of the await expression. Use of async and await enables the use of ordinary try / catch blocks around asynchronous code.

**Note:** The await keyword is only valid inside async functions within regular JavaScript code. If you use it outside of an async function's body, you will get a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError).

await can be used on its own with [JavaScript modules.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules)

**Note:** The purpose of async/await is to simplify the syntax necessary to consume promise-based APIs. The behavior of async/await is similar to combining [generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_generators) and promises.

Async functions always return a promise. If the return value of an async function is not explicitly a promise, it will be implicitly wrapped in a promise.

For example, consider the following code:

JSCopy to Clipboard

async function foo() {

return 1;

}

It is similar to:

JSCopy to Clipboard

function foo() {

return Promise.resolve(1);

}

**Note:**

Even though the return value of an async function behaves as if it's wrapped in a Promise.resolve, they are not equivalent.

An async function will return a different *reference*, whereas Promise.resolve returns the same reference if the given value is a promise.

It can be a problem when you want to check the equality of a promise and a return value of an async function.

JSCopy to Clipboard

const p = new Promise((res, rej) => {

res(1);

});

async function asyncReturn() {

return p;

}

function basicReturn() {

return Promise.resolve(p);

}

console.log(p === basicReturn()); // true

console.log(p === asyncReturn()); // false

The body of an async function can be thought of as being split by zero or more await expressions. Top-level code, up to and including the first await expression (if there is one), is run synchronously. In this way, an async function without an await expression will run synchronously. If there is an await expression inside the function body, however, the async function will always complete asynchronously.

For example:

JSCopy to Clipboard

async function foo() {

await 1;

}

It is also equivalent to:

JSCopy to Clipboard

function foo() {

return Promise.resolve(1).then(() => undefined);

}

Code after each await expression can be thought of as existing in a .then callback. In this way a promise chain is progressively constructed with each reentrant step through the function. The return value forms the final link in the chain.

In the following example, we successively await two promises. Progress moves through function foo in three stages.

1. The first line of the body of function foo is executed synchronously, with the await expression configured with the pending promise. Progress through foo is then suspended and control is yielded back to the function that called foo.
2. Some time later, when the first promise has either been fulfilled or rejected, control moves back into foo. The result of the first promise fulfillment (if it was not rejected) is returned from the await expression. Here 1 is assigned to result1. Progress continues, and the second await expression is evaluated. Again, progress through foo is suspended and control is yielded.
3. Some time later, when the second promise has either been fulfilled or rejected, control re-enters foo. The result of the second promise resolution is returned from the second await expression. Here 2 is assigned to result2. Control moves to the return expression (if any). The default return value of undefined is returned as the resolution value of the current promise.

JSCopy to Clipboard

async function foo() {

const result1 = await new Promise((resolve) =>

setTimeout(() => resolve("1")),

);

const result2 = await new Promise((resolve) =>

setTimeout(() => resolve("2")),

);

}

foo();

Note how the promise chain is not built-up in one go. Instead, the promise chain is constructed in stages as control is successively yielded from and returned to the async function. As a result, we must be mindful of error handling behavior when dealing with concurrent asynchronous operations.

For example, in the following code an unhandled promise rejection error will be thrown, even if a .catch handler has been configured further along the promise chain. This is because p2 will not be "wired into" the promise chain until control returns from p1.

JSCopy to Clipboard

async function foo() {

const p1 = new Promise((resolve) => setTimeout(() => resolve("1"), 1000));

const p2 = new Promise((\_, reject) => setTimeout(() => reject("2"), 500));

const results = [await p1, await p2]; // Do not do this! Use Promise.all or Promise.allSettled instead.

}

foo().catch(() => {}); // Attempt to swallow all errors...

async function declarations behave similar to [function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function) declarations — they are [hoisted](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting) to the top of their scope and can be called anywhere in their scope, and they can be redeclared only in certain contexts.

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#examples)

[**Async functions and execution order**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function#async_functions_and_execution_order)

JSCopy to Clipboard

function resolveAfter2Seconds() {

console.log("starting slow promise");

return new Promise((resolve) => {

setTimeout(() => {

resolve("slow");

console.log("slow promise is done");

}, 2000);

});

}

function resolveAfter1Second() {

console.log("starting fast promise");

return new Promise((resolve) => {

setTimeout(() => {

resolve("fast");

console.log("fast promise is done");

}, 1000);

});

}

async function sequentialStart() {

console.log("== sequentialStart starts ==");

// 1. Start a timer, log after it's done

const slow = resolveAfter2Seconds();

console.log(await slow);

// 2. Start the next timer after waiting for the previous one

const fast = resolveAfter1Second();

console.log(await fast);

console.log("== sequentialStart done ==");

}

async function sequentialWait() {

console.log("== sequentialWait starts ==");

// 1. Start two timers without waiting for each other

const slow = resolveAfter2Seconds();

const fast = resolveAfter1Second();

// 2. Wait for the slow timer to complete, and then log the result

console.log(await slow);

// 3. Wait for the fast timer to complete, and then log the result

console.log(await fast);

console.log("== sequentialWait done ==");

}

async function concurrent1() {

console.log("== concurrent1 starts ==");

// 1. Start two timers concurrently and wait for both to complete

const results = await Promise.all([

resolveAfter2Seconds(),

resolveAfter1Second(),

]);

// 2. Log the results together

console.log(results[0]);

console.log(results[1]);

console.log("== concurrent1 done ==");

}

async function concurrent2() {

console.log("== concurrent2 starts ==");

// 1. Start two timers concurrently, log immediately after each one is done

await Promise.all([

(async () => console.log(await resolveAfter2Seconds()))(),

(async () => console.log(await resolveAfter1Second()))(),

]);

console.log("== concurrent2 done ==");

}

sequentialStart(); // after 2 seconds, logs "slow", then after 1 more second, "fast"

// wait above to finish

setTimeout(sequentialWait, 4000); // after 2 seconds, logs "slow" and then "fast"

// wait again

setTimeout(concurrent1, 7000); // same as sequentialWait

// wait again

setTimeout(concurrent2, 10000); // after 1 second, logs "fast", then after 1 more second, "slow"

**await and concurrency**

In sequentialStart, execution suspends 2 seconds for the first await, and then another second for the second await. The second timer is not created until the first has already fired, so the code finishes after 3 seconds.

In sequentialWait, both timers are created and then awaited. The timers run concurrently, which means the code finishes in 2 rather than 3 seconds, i.e. the slowest timer. However, the await calls still run in series, which means the second await will wait for the first one to finish. In this case, the result of the fastest timer is processed after the slowest.

If you wish to safely perform other jobs after two or more jobs run concurrently and are complete, you must await a call to [Promise.all()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all) or [Promise.allSettled()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled) before that job.