**Async/await**

There’s a special syntax to work with promises in a more comfortable fashion, called “async/await”. It’s surprisingly easy to understand and use.

**[Async functions](https://javascript.info/async-await" \l "async-functions)**

Let’s start with the async keyword. It can be placed before a function, like this:

async function f() {

return 1;

}

The word “async” before a function means one simple thing: a function always returns a promise. If the code has return <non-promise> in it, then JavaScript automatically wraps it into a resolved promise with that value.

For instance, the code above returns a resolved promise with the result of 1, let’s test it:

async function f() {

return 1;

}

f().then(alert); // 1

…We could explicitly return a promise, that would be the same:

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

So, async ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There’s another keyword, await, that works only inside async functions, and it’s pretty cool.

**[Await](https://javascript.info/async-await" \l "await)**

The syntax:

// works only inside async functions

let value = await promise;

The keyword await makes JavaScript wait until that promise settles and returns its result.

Here’s an example with a promise that resolves in 1 second:

async function f() {

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

setTimeout(() => resolve("done!"), 1000)

});

let result = await promise; // wait till the promise resolves (\*)

alert(result); // "done!"

}

f();

The function execution “pauses” at the line (\*) and resumes when the promise settles, with result becoming its result. So the code above shows “done!” in one second.

Let’s emphasize: await literally makes JavaScript wait until the promise settles, and then go on with the result. That doesn’t cost any CPU resources, because the engine can do other jobs meanwhile: execute other scripts, handle events etc.

It’s just a more elegant syntax of getting the promise result than promise.then, easier to read and write.

**Can’t use await in regular functions**

If we try to use await in non-async function, that would be a syntax error:

function f() {

let promise = Promise.resolve(1);

let result = await promise; // Syntax error

}

We can get such error in case if we forget to put async before a function. As said, await only works inside async function.

Let’s take showAvatar() example from the chapter [Promises chaining](https://javascript.info/promise-chaining) and rewrite it using async/await:

1. We’ll need to replace .then calls by await.
2. Also we should make the function async for them to work.

async function showAvatar() {

// read our JSON

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

// read github user

let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);

let githubUser = await githubResponse.json();

// show the avatar

let img = document.createElement('img');

img.src = githubUser.avatar\_url;

img.className = "promise-avatar-example";

document.body.append(img);

// wait 3 seconds

await new Promise((resolve, reject) => setTimeout(resolve, 3000));

img.remove();

return githubUser;

}

showAvatar();

Pretty clean and easy to read, right? Much better than before.

**await won’t work in the top-level code**

People who are just starting to use await tend to forget that, but we can’t write await in the top-level code. That wouldn’t work:

// syntax error in top-level code

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

So we need to have a wrapping async function for the code that awaits. Just as in the example above.

**await accepts thenables**

Like promise.then, await allows to use thenable objects (those with a callable then method). Again, the idea is that a 3rd-party object may not be a promise, but promise-compatible: if it supports .then, that’s enough to use with await.

For instance, here await accepts new Thenable(1):

class Thenable {

constructor(num) {

this.num = num;

}

then(resolve, reject) {

alert(resolve); // function() { native code }

// resolve with this.num\*2 after 1000ms

setTimeout(() => resolve(this.num \* 2), 1000); // (\*)

}

};

async function f() {

// waits for 1 second, then result becomes 2

let result = await new Thenable(1);

alert(result);

}

f();

If await gets a non-promise object with .then, it calls that method providing native functions resolve, reject as arguments. Then await waits until one of them is called (in the example above it happens in the line (\*)) and then proceeds with the result.