# Async/await

In computer programming, the **async/await pattern** is a syntactic feature of many programming languages that allows an asynchronous, non-blocking function to be structured in a way similar to an ordinary synchronous function. It is semantically related to the concept of a coroutine and is often implemented using similar techniques, but is primarily intended to provide opportunities for the program to execute other code while waiting for a long-running, asynchronous task to complete, usually represented by promises or similar data structures. The feature is found in C# 5.0, Python 3.5, Hack, Dart, Kotlin 1.1, and JavaScript, with some experimental work in extensions, beta versions, and particular implementations of Scala[1], Rust[2], and C++.

## Contents



## Example C#

The C# function below, which downloads a resource from a URI and returns the resource's length, uses this async/await pattern:

```
public async Task<int> FindPageSize(Uri uri)
{
    byte[] data = await new WebClient().DownloadDataTaskAsync(uri);
    return data.Length;
}
```

- First, the `async` keyword indicates to C# that the method is asynchronous, meaning that it will use one or more `await` expressions and will bind the result to a promise.
- The return type, `Task<T>`, is C#'s analogue to the concept of a promise, and here is indicated to have a result value of type `int`.
- The first expression to execute when this method is called will be new `WebClient().DownloadDataTaskAsync(uri)`, which is another asynchronous method returning a `Task<byte[]>`. Because this method is asynchronous, it will not download the entire batch of data before returning. Instead, it will begin the download process using a non-blocking mechanism (such as a background thread), and immediately return an unresolved, unrejected `Task<byte[]>` to this function.
- With the `await` keyword attached to the Task, this function will immediately proceed to return a `Task<int>` to its caller, who may then continue on with other processing as needed.
- Once `DownloadDataTaskAsync()` finishes its download, it will resolve the `Task` it returned with the downloaded data. This will trigger a callback and cause `FindPageSize()` to continue execution by assigning that value to `data`.
- Finally, the method returns `data.Length`, a simple integer indicating the length of the array. The compiler re-interprets this as resolving the `Task` it returned earlier, triggering a callback in the method's caller to do something with that length value.

A function using async/await can use as many `await` expressions as it wants, and each will be handled in the same way (though a promise will only be returned to the caller for the first await, while every other await will utilize internal callbacks). A function can also hold a promise object directly and do other processing first (including starting other asynchronous tasks), delaying awaiting the promise until its result is needed. Functions with promises also have promise aggregation methods that allow you to await multiple promises at once or in some special pattern (such as C#'s `Task.WhenAll()`, which returns a valueless `Task` that resolves when all of the tasks in the arguments have resolved). Many promise types also have additional features beyond what the async/await pattern normally uses, such as being able to set up more than one result callback or inspect the progress of an especially long-running task.

In the particular case of C#, and in many other languages with this language feature, the async/await pattern is not a core part of the language's runtime, but is instead implemented with lambdas or continuations at compile time. For instance, the C# compiler would likely translate the above code to something like the following before translating it to its IL bytecode format:

```
public Task<int> FindPageSize(Uri uri)
{
    Task<byte[]> data_task = new WebClient().DownloadDataTaskAsync(uri);
    Task<int> after_data_task = data_task.ContinueWith((original_task) => {
        return original_task.Result.Length;
    });
    return after_data_task;
}
```

Because of this, if an interface method needs to return a promise object, but itself does not require `await` in the body to wait on any asynchronous tasks, it does not need the `async` modifier either and can instead return a promise object directly. For instance, a function might be able to provide a promise that immediately resolves to some result value (such as C#'s `Task.FromResult()`), or it may simply return another method's promise that happens to be the exact promise needed (such as when deferring to an overload).

One important caveat of this functionality, however, is that while the code resembles traditional blocking code, the code is actually non-blocking and potentially multithreaded, meaning that many intervening events may occur while waiting for the promise targeted by an `await` to resolve. For instance, the following code, while always succeeding in a blocking model without `await`, may experience intervening events during the `await` and may thus find shared state changed out from under it:

```
var a = state.a;
var data = await new WebClient().DownloadDataTaskAsync(uri);
Debug.Assert(a == state.a);//potential failure, as value of state.a may have been changed
                           //  by the handler of potentially intervening event
return data.Length;
```

## In F#

An F# release of 2007 featured *asynchronous workflows*.[3]. In this initial version, `await` was called `let!`.

## In C#

In C# versions before C# 7, async methods are required to return either `void`, `Task`, or `Task<T>`. This has been expanded in C# 7 to include certain other types such as `ValueTask<T>`. Async methods that return `void` are intended for event handlers; in most cases where a synchronous method would return `void`, returning `Task` instead is recommended, as it allows for more intuitive exception handling.[4]

Methods that make use of `await` must be declared with the `async` keyword. In methods that have a return value of type `Task<T>`, methods declared with `async` must have a return statement of type assignable to `T` instead of `Task<T>`; the compiler wraps the value in the `Task<T>` generic. It is also possible to `await` methods that have a return type of `Task` or `Task<T>` that are declared without `async`.

The following async method downloads data from a URL using `await`.

```
public async Task<int> SumPageSizesAsync(IList<Uri> uris)
{
    int total = 0;
    foreach (var uri in uris) {
        statusText.Text = string.Format("Found {0} bytes ...", total);
        var data = await new WebClient().DownloadDataTaskAsync(uri);
        total += data.Length;
    }
    statusText.Text = string.Format("Found {0} bytes total", total);
    return total;
}
```

## In Scala

In the experimental Scala-async extension to Scala, `await` is a "method", although it does not operate like an ordinary method. Furthermore, unlike in C# 5.0 in which a method must be marked as async, in Scala-async, a *block* of code is surrounded by an async "call".

### How it works

In Scala-async, `async` is actually implemented using a Scala macro, which causes the compiler to emit different code, and produce a finite state machine implementation (which is considered to be more efficient than a monadic implementation, but less convenient to write by hand).

There are plans for Scala-async to support a variety of different implementations, including non-asynchronous ones.

## In Python

Python 3.5 has added support for Async/Await as described in PEP0492 (https://www.python.org/dev/peps/pep-0492/).

## In JavaScript

The await operator in JavaScript can only be used from inside an async function. If the parameter is a promise, execution of the async function will resume when the promise is resolved (unless the promise is rejected, in which case an error will be thrown that can be handled with normal JavaScript exception handling.) If the parameter is not a promise, the parameter itself will be returned immediately.[5]

Many libraries provide promise objects that can also be used with await, as long as they match the specification for native JavaScript promises. However, promises from the jQuery library were not Promises/A+ compatible until jQuery 3.0.[6]

Here's an example (modified from this[7] article):

```
async function createNewDoc() {
  let response = await db.post({}); // post a new doc
  return await db.get(response.id); // find by id
}

async function main() {
  try {
    let doc = await createNewDoc();
    console.log(doc);
  } catch (err) {
    console.log(err);
  }
}()
```

Node.js version 8 includes a utility that enables using the standard library callback-based methods as promises.[8]

Async functions always return a promise. If the coder explicitly returns a value at the end of the async function, the promise will be resolved with that value; otherwise, it resolves with `undefined`.[9] This means async functions can be chained like pure promise based functions.

## In C++

In C++, await is a part of Coroutines TS, and uses the keyword co_await. Coroutines TS may or may not be merged into C++20, [10] but MSVC and Clang compilers are already supporting at least some form of co_await (GCC has still no support for it).

## Benefits and criticisms

A significant benefit of the async/await pattern in languages that support it is that asynchronous, non-blocking code can be written, with minimal overhead, and looking almost like traditional synchronous, blocking code. In particular, it has been argued that await is the best way of writing asynchronous code in message-passing programs; in particular, being close to blocking code, readability and the minimal amount of boilerplate code were cited as await benefits.[11] As a result, async/await makes it easier for most programmers to reason about their programs, and await tends to promote better, more robust non-blocking code in applications that require it. Such applications range from programs presenting graphical user interfaces to massively scalable stateful server-side programs, such as games and financial applications.

When criticising await, it has been noted that await tends to cause surrounding code to be asynchronous too; on the other hand, it has been argued that this contagious nature of the code (sometimes being compared to a "zombie virus") is inherent to all kinds of asynchronous programming, so await as such is not unique in this regard.[4]

## See also

- Coroutines
- Continuation-passing style
- Direct style
- Cooperative multitasking

## References


1. "Scala Async" (https://github.com/scala/async). Retrieved 20 October 2013.
2. "alexcrichton/futures-await" (https://github.com/alexcrichton/futures-await). *GitHub*. Retrieved 2018-03-29.
3. "Introducing F# Asynchronous Workflows" (https://blogs.msdn.microsoft.com/dsyme/2007/10/10/introducing-f-asynchronous-workflows/).
4. Stephen Cleary, Async/Await - Best Practices in Asynchronous Programming (https://msdn.microsoft.com/en-us/magazine/jj991977.aspx)
5. "await - JavaScript (MDN)" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await). Retrieved 2 May 2017.
6. "jQuery Core 3.0 Upgrade Guide" (https://jquery.com/upgrade-guide/3.0/#breaking-change-and-feature-jquery-deferred-is-now-promises-a-compatible). Retrieved 2 May 2017.
7. "Taming the asynchronous beast with ES7" (http://pouchdb.com/2015/03/05/taming-the-async-beast-with-es7.html). Retrieved 12 November 2015.
8. Foundation, Node.js. "Node v8.0.0 (Current) - Node.js" (https://nodejs.org/en/blog/release/v8.0.0/#improved-support-for-promises). *Node.js*.
9. Chiang, George. "Introduction to JavaScript Async Functions- Promises simplified" (http://www.javascriptkit.com/javatutors/intro-javascript-async-functions.shtml).
10. "Trip report: Fall ISO C++ standards meeting (San Diego)" (https://herbsutter.com/2018/11/13/trip-report-fall-iso-c-standards-meeting-san-diego/). 13 Nov 2018.
11. 'No Bugs' Hare. Eight ways to handle non-blocking returns in message-passing programs (http://ithare.com/eight-ways-to-handle-non-blocking-returns-in-message-passing-programs-with-script/) CPPCON, 2018