**Async Await**

This is a syntactic sugar, a different form of using promises and generators. The async...await syntax allows us to write asynchronous code that reads similarly to traditional synchronous, imperative programs.

1. **The async keyword**

The async keyword is used to write functions that handle asynchronous actions. There are 2 ways:

Text

Description automatically generatedText

Description automatically generated

async functions always return a promise. This means we can use traditional promise syntax, like .then() and .catch() with our async functions. An async function will return in one of three ways:

* If there’s nothing returned from the function, it will return a promise with a resolved value of undefined.
* If there’s a non-promise value returned from the function, it will return a promise resolved to that value.
* If a promise is returned from the function, it will simply return that promise

1. **The await Operator**

The await keyword can only be used inside an async function. await is an operator

await **returns the resolved value** of a promise (not the settled promise 🡪 no need for .then() afterward)

Since promises resolve in an indeterminate amount of time, await pauses the execution of our async function until a given promise is resolved, making sure that the next actions will only be done after the promise has been settled (or else, it will return [object Promise], which is not logic at all).

Text

Description automatically generatedText

Description automatically generated

1. **Handling Dependent Promises**

The true beauty of async...await is when we have a series of asynchronous actions which depend on one another. For example, we may make a network request based on a query to a database. In that case, we would need to wait to make the network request until we had the results from the database.

Text

Description automatically generated

🡪 This is more easy to store and refer to resolved values from promises, which is a much more difficult task with native promise syntax.

1. **Handling Errors**

Previous examples ALL do not have rejected values. When there are rejected values from Promises, we MUST handle them with .catch() with native promises or try…catch

When .catch() is used with a long promise chain, there is no indication of where in the chain the error was thrown. This can make debugging challenging.

With async...await, we use **try...catch** statements for error handling. By using this syntax, not only are we able to handle errors in the same way we do with synchronous code, but we can also catch both synchronous and asynchronous errors.

Text

Description automatically generated

*(note that this can still be done with .catch() as normal, in the bottom of the code.*

1. **Handling Independent Promises**

What if our async function contains multiple promises which are not dependent on the results of one another to execute?

Text

Description automatically generatedIn waiting(), the function is paused until first promise resolves, then we construct second promise. Once this resolves, we log them.

On the other hand, in concurrent(), both promises are constructed without using await. We then await each of their resolutions to print them to the console.

🡪 In concurrent, both promises’ asynchronous operations can be run simultaneously.

***Note****: if we have multiple truly independent promises that we would like to execute fully in parallel, we must use individual .then() functions and avoid halting our execution with await.*

1. **Await Promise.all()**

Another way to take advantage of concurrency when we have multiple promises which can be executed simultaneously is to await a Promise.all().

We can pass an array of promises as the argument to Promise.all(), and it will return a single promise. This promise will resolve when all of the promises in the argument array have resolved. This promise’s resolve value will be an array containing the resolved values of each promise from the argument array.

Text

Description automatically generated

Promise.all() allows us to take advantage of asynchronicity— each of the four asynchronous tasks can process concurrently. Promise.all() also has the benefit of *failing fast.* As soon as the first promise in the array rejects, the promise returned from Promise.all() will reject with that reason.