Chapter 81: Async functions (async/await)

async and await build on top of promises and generators to express asynchronous actions inline. This makes asynchronous or callback code much easier to maintain.

Functions with the async keyword return a Promise, and can be called with that syntax.

Inside an async function the await keyword can be applied to any Promise, and will cause all of the function body after the await to be executed after the promise resolves.

Section 81.1: Introduction

A function defined as async is a function that can perform asynchronous actions but still look synchronous. The way it's done is using the await keyword to defer the function while it waits for a Promise to resolve or reject.

Note: Async functions are a Stage 4 ("Finished") proposal on track to be included in the ECMAScript 2017 standard.

For instance, using the promise-based Fetch API:

```
async function getJSON(url) {
    try {
        const response = await fetch(url);
        return await response.json();
    }
    catch (err) {
        // Rejections in the promise will get thrown here
        console.error(err.message);
    }
}
```

An async function always returns a Promise itself, so you can use it in other asynchronous functions.

Arrow function style

```
const getJSON = async url => {
    const response = await fetch(url);
    return await response.json();
}
```

Section 81.2: Await and operator precedence

You have to keep the operator precedence in mind when using await keyword.

Imagine that we have an asynchronous function which calls another asynchronous function, getUnicorn() which returns a Promise that resolves to an instance of class Unicorn. Now we want to get the size of the unicorn using the getSize() method of that class.

Look at the following code:

```
async function myAsyncFunction() {
    await getUnicorn().getSize();
}
```

At first sight, it seems valid, but it's not. Due to operator precedence, it's equivalent to the following:

```
async function myAsyncFunction() {
    await (getUnicorn().getSize());
}
```

Here we attempt to call `getSize()` method of the Promise object, which isn't what we want.

Instead, we should use brackets to denote that we first want to wait for the unicorn, and then call `getSize()` method of the result:

```
async function asyncFunction() {
    (await getUnicorn()).getSize();
}
```

Of course. the previous version could be valid in some cases, for example, if the `getUnicorn()` function was synchronous, but the `getSize()` method was asynchronous.

Section 81.3: Async functions compared to Promises

`async` functions do not replace the `Promise` type; they add language keywords that make promises easier to call. They are interchangeable:

```
async function doAsyncThing() { ... }

function doPromiseThing(input) { return new Promise((r, x) => ...); }

// Call with promise syntax
doAsyncThing()
    .then(a => doPromiseThing(a))
    .then(b => ...)
    .catch(ex => ...);

// Call with await syntax
try {
    const a = await doAsyncThing();
    const b = await doPromiseThing(a);
    ...
}
catch(ex) { ... }
```

Any function that uses chains of promises can be rewritten using `await`:

```
function newUnicorn() {
  return fetch('unicorn.json')                     // fetch unicorn.json from server
  .then(responseCurrent => responseCurrent.json()) // parse the response as JSON
  .then(unicorn =>
    fetch('new/unicorn', {                         // send a request to 'new/unicorn'
        method: 'post',                            // using the POST method
        body: JSON.stringify({unicorn})            // pass the unicorn to the request body
    })
  )
  .then(responseNew => responseNew.json())
  .then(json => json.success)                      // return success property of response
  .catch(err => console.log('Error creating unicorn:', err));
 }
```

The function can be rewritten using `async` / `await` as follows:

```
async function newUnicorn() {
  try {
    const responseCurrent = await fetch('unicorn.json'); // fetch unicorn.json from server
    const unicorn = await responseCurrent.json();        // parse the response as JSON
    const responseNew = await fetch('new/unicorn', {     // send a request to 'new/unicorn'
      method: 'post',                                    // using the POST method
      body: JSON.stringify({unicorn})                    // pass the unicorn to the request body
    });
    const json = await responseNew.json();
    return json.success                                  // return success property of response
  } catch (err) {
    console.log('Error creating unicorn:', err);
  }
}
```

This `async` variant of `newUnicorn()` appears to return a `Promise`, but really there were multiple `await` keywords. Each one returned a `Promise`, so really we had a collection of promises rather than a chain.

In fact we can think of it as a `function*` generator, with each `await` being a `yield new Promise`. However, the results of each promise are needed by the next to continue the function. This is why the additional keyword `async` is needed on the function (as well as the `await` keyword when calling the promises) as it tells JavaScript to automatically creates an observer for this iteration. The `Promise` returned by `async function newUnicorn()` resolves when this iteration completes.

Practically, you don't need to consider that; `await` hides the promise and `async` hides the generator iteration.

You can call `async` functions as if they were promises, and `await` any promise or any `async` function. You don't need to `await` an async function, just as you can execute a promise without a `.then()`.

You can also use an `async` IIFE if you want to execute that code immediately:

```
(async () => {
  await makeCoffee()
  console.log('coffee is ready!')
})()
```

Section 81.4: Looping with async await

When using async await in loops, you might encounter some of these problems.

If you just try to use await inside `forEach`, this will throw an `Unexpected token` error.

```
(async() => {
 data = [1, 2, 3, 4, 5];
 data.forEach(e => {
   const i = await somePromiseFn(e);
   console.log(i);
 });
})();
```

This comes from the fact that you've erroneously seen the arrow function as a block. The `await` will be in the context of the callback function, which is not `async`.
The interpreter protects us from making the above error, but if you add `async` to the `forEach` callback no errors get thrown. You might think this solves the problem, but it won't work as expected.

Example:

```
(async() => {
  data = [1, 2, 3, 4, 5];
  data.forEach(async(e) => {
    const i = await somePromiseFn(e);
    console.log(i);
  });
  console.log('this will print first');
})();
```

This happens because the callback async function can only pause itself, not the parent async function.

> You could write an asyncForEach function that returns a promise and then you could something like `await asyncForEach(async (e) => await somePromiseFn(e), data )` Basically you return a promise that resolves when all the callbacks are awaited and done. But there are better ways of doing this, and that is to just use a loop.

You can use a `for-of` loop or a `for/while` loop, it doesn't really matter which one you pick.

```
(async() => {
  data = [1, 2, 3, 4, 5];
  for (let e of data) {
    const i = await somePromiseFn(e);
    console.log(i);
  }
  console.log('this will print last');
})();
```

But there's another catch. This solution will wait for each call to `somePromiseFn` to complete before iterating over the next one.
This is great if you actually want your `somePromiseFn` invocations to be executed in order but if you want them to run concurrently, you will need to `await` on `Promise.all`.

```
(async() => {
 data = [1, 2, 3, 4, 5];
 const p = await Promise.all(data.map(async(e) => await somePromiseFn(e)));
 console.log(...p);
})();
```

`Promise.all` receives an array of promises as its only parameter and returns a promise. When all of the promises in the array are resolved, the returned promise is also resolved. We `await` on that promise and when it's resolved all our values are available.

The above examples are fully runnable. The `somePromiseFn` function can be made as an async echo function with a timeout. You can try out the examples in the babel-repl with at least the `stage-3` preset and look at the output.

```
function somePromiseFn(n) {
 return new Promise((res, rej) => {
   setTimeout(() => res(n), 250);
 });
}
```

Section 81.5: Less indentation

With promises:

```
function doTheThing() {
    return doOneThing()
```

```
        .then(doAnother)
        .then(doSomeMore)
        .catch(handleErrors)
}
```

With async functions:

```
async function doTheThing() {
    try {
        const one = await doOneThing();
        const another = await doAnother(one);
        return await doSomeMore(another);
    } catch (err) {
        handleErrors(err);
    }
}
```

Note how the return is at the bottom, and not at the top, and you use the language's native error-handling mechanics (`try/catch`).

Section 81.6: Simultaneous async (parallel) operations

Often you will want to perform asynchronous operations in parallel. There is direct syntax that supports this in the `async/await` proposal, but since `await` will wait for a promise, you can wrap multiple promises together in `Promise.all` to wait for them:

```
// Not in parallel

async function getFriendPosts(user) {
    friendIds = await db.get("friends", {user}, {id: 1});
    friendPosts = [];
    for (let id in friendIds) {
        friendPosts = friendPosts.concat( await db.get("posts", {user: id}) );
    }
    // etc.
}
```

This will do each query to get each friend's posts serially, but they can be done simultaneously:

```
// In parallel

async function getFriendPosts(user) {
    friendIds = await.db.get("friends", {user}, {id: 1});
    friendPosts = await Promise.all( friendIds.map(id =>
      db.get("posts", {user: id})
    );
    // etc.
}
```

This will loop over the list of IDs to create an array of promises. `await` will wait for *all* promises to be complete. `Promise.all` combines them into a single promise, but they are done in parallel.