Lesson 2.1 - Introducing Modules

The first thing we're going to talk about in this course is the introduction of a standard way to encapsulate code using modules.

Up until the adoption of ES2015, JavaScript did not have any sort of defined module system. Both RequireJS (an AMD style module loading system) and CommonJS (a synchronous, blocking loading system) stepped in to try to solve this problem years ago, and have achieved wide adoption amongst developers, but there was always still contention as to which to use.

ES2015 style modules have yet to be implemented in any browser or NodeJS. We're using Babel here to be able to use the new syntax in our code. By default Babel will transpile your code to use the CommonJS style module syntax, and thus making it synchronous and blocking. You can change this behavior by defining a different plugin in the `.babelrc` file, but for the purposes of this course, we will be using the default behavior.

Let's see how this works!

Module Exporting

There are essentially two distinct ways to export code from a module and expose it to the world. The first thing to do is to simply export a variable or function:

```
export function add(x, y) {
  return x + y;
}

export const foo = 1;
```

Requiring this module in another file will give you an immutable binding. This differs from CommonJS to make your code more deterministic and circular dependencies easier to work with. A transpiler can only do so much so for our purposes the exported binding will look like an object literal like this:

```
{
  add: function add(x, y) { ... },
  foo: 1
}
```

You can then just refer to the export that you are looking for by the name of the key! This is great, but often times you will want to only export one thing from a module, and have that be the default export. This is where the, you guessed it, `default` keyword comes in:

```
export default function add(x, y) {
  return x + y;
}
```

Now, if you import that, you can refer to the name of the function directly and import it all in one step.

Module Importing

Importing modules works very much the same way as exporting. Let's refer back to our last example where we exported the add function as the default. In another module, we can import that like this:

```
import add from './my_math_module';
```

Now we will have the add function available in our file! It's that easy! You can also use named imports to import multiple exports all at once. If we change our module back to it's original form like this:

```
export function add(x, y) {
  return x + y;
}

export const foo = 1;
```

We can then import both of these items like so:

```
import { add, foo } from './my_math_module';
```

A word on destructuring

This named import syntax: `{ add, foo }` may look a little strange to you, but don't worry! We will be convering this in detail in a later lesson. Essentially what is happening here is we are saying that we only want to import specific pieces from the module. Here we are only importing the add function and the `foo` variable.

More Advanced Module Syntax

In addition to these basic statements, we can import and export in a few different ways as well.

There may be times when you want to export a module from *another* module. In this case you can do this directly:

```
export add from './my_math_module';
```

Nifty eh? How about exporting from a module that doesn't have a default? Yeah, we can do that too!

```
export * from './my_math_module';
```

or

```
export { add } from './my_math_module';
```

How about exporting from a module and changing the name? Yeah we can do that too:

```
export * as math from './my_math_module';
```

We can use most of the same syntax for importing modules as well!

```
import * as math from './my_math_module';
```

You can also import the default as well as other exports. Let's say our module looked like this:

```
export default function add(x, y) {
  return x + y;
}

export const foo = 1;
```

You can then import both of those like this:

```
import add, { foo } from './my_math_module';
```

What's that? You want to see more? Ok, how about named imports and exports!

```
import add as a from './my_math_module';
export add as a from './my_math_module';
```