## Modules

**Module Loading**

By default javascript doesn’t have a module system like other languages, e.g. Java or Python.

This means that if you wanted to call a function in some other file, you have to remember to *explicitly* load that file via script tags *before* you call the function.

If you tried to use code that you forgot to add via a script tag, then javascript would complain.

Other languages have a module loading system e.g. in python if you wanted to use some code from another file you would type something like

import foo from bar;

foo();

The language itself figured out *where* bar was, loaded it up from the filesystem, extracted the function foo and made it available to you in your file to use.

This feature was missing in JavaScript so the community developed their own solutions, such as *CommonJS* which is used in node.

**ES6 Modules**

ES6 took the best of the existing module systems and introduced this concept on a language level.

Although it’s made it into the ES6 standard it’s up to the javascript engine makers to *actually* implement it natively and they *haven’t*… *yet*.

So until that happens we code using the ES6 module syntax in TypeScript. When typescript transpiles the code to ES5 it uses the CommonJS module loading system which we touched on above.

 We can configure TypeScript to use other module loaders, but the default isCommonJS.

**Exporting**

// utils.ts

function square(x) {

return Math.pow(x,2)

}

function cow() {

console.log("Mooooo!!!")

}

export {square, cow};

We declare some functions in a file.

By using the export keyword we say *which* of those functions can be exported, and therefore imported and used in other modules.

  cow}{square, cow. } is just destructuring syntax and is short for {square: square, cow:

**Importing**

// script.ts

import {square, cow} from './utils';

console.log(square(2));

cow();



To compile with typescript we need to provide both files on the command line

tsc

-

t ES5 -w utils.ts script.ts

We again use that destructuring syntax to import the functions we want from the utils module, we provide a path relative to *this* module.

**Aliases**

We may want to import a function with one name but then use it via another name. Perhaps to avoid name collisions or just to have a more convenient naming, like so:

import {square as sqr} from './utils';

sqr(2);

Or we can import everything in a file like so:

import \* as utils from './utils';

console.log(utils.square(2));

utils.cow();

**Alternative export syntax**

As well as describing the exports by using the export keyword, like so:

export {square, cow};

We can also export functions or variables as they are defined by prepended the word export to the front of their declarations:

export function square(x) {

return Math.pow(x,2)

}

## Default exports

If a module defines one export which is the most common, we can take advantage of the *default export* syntax, like so:

export default function square(x) {

return Math.pow(x,2)

}

And then when we import it we don’t need to use { }, like so:

import square from './utils';

Or, if we want to import the default export as well as some other exports, we can use:

import square, { cow } from './utils';

**Summary**

With ES6 modules we finally have a mechanism for letting the language deal with loading of dependant files for us.

This isn’t baked into javascript engines yet. So to solve this problem in Angular we still use the ES6 module loading syntax but leave it to TypeScript to transpile to CommonJS.

**Listing**

<https://github.com/codecraftpro/angular2-sample-code/tree/master/2.es6-typescript/12.modules>

*script.js*

import \* as utils from './utils';

console.log(utils.square(4));

utils.cow();

*utils.js*

export function square(x) {

return Math.pow(x, 2)

}

export function cow() {

console.log("Mooooo!!!")

}