Overview

- Need for modules.
- Do-it-yourself modules.
- Commonjs modules.
- Asynchronous modules.
- Concentrate on ES6 modules.

Need for Modules

- Tom writes a file `math.js` which includes a top-level `sin()` function.
- Jill writes a file `morals.js` which includes a top-level `sin()` function.
- How does program reuse the code from both files?
- This is a problem with *programming in the large*.
- Common solution is to have programming language support modules; for example Java has packages and modules (since Java 9); C++ has namespaces.
- Javascript had no support for modules until ES6.

DIY Modules: IIFEs

Code can be loaded and run in a browser using code like the following:

```
(function () {
  const constant = ...;
  let var = ...;
  function f1(element) { ... var ... constant ... }
  function f2(element) { ... f1(); ... var ... }

  //code which is run on current browser document
  f2(document.getElementById('shapes'));
})();
```

DIY Modules: Module Revealing Pattern

```
const math = (function() {
  function abs() { ... }
  ...
  function sin() { ... }
  ...
  return { abs, ..., sin, ... };
})();

const morals = (function() {
  function doGood() { ... }
  ...
  function sin() { ... }
  ...
  return { doGood, ..., sin, ... };
})();
```

Module Revealing Pattern Continued

```
let x = ...;
math.sin(Math.PI/4);   //ok
morals.sin('no more'); //ok
math.abs(x*3);          //ok
morals.abs(...);        //error
math.doGood();           //error
```

Different Startup Needs between Browser and Server

JavaScript can only be doing one thing at a time:

- When a web page is loaded into a browser:
  - It may load multiple remote scripts and other resources.
  - If scripts are loaded **synchronously**, then browser will block during loading; this will result in unresponsive web pages.
  - Hence for a browser, external resources must be loaded **asynchronously**.
  - Resulted in **Asynchronous Module Definitions** AMD for use in browsers.
  - Circular dependencies problematic.
- When a server-side application is started up, perfectly acceptable to wait for resources to be loaded into application. Hence **synchronous** loading is acceptable.
  - CommonJS specification (importing done using `require()` function).
  - Handles circular dependencies.
  - Emulated for browsers using server-side packaging tools like webpack.

ES6 Modules

- Distinguish between JavaScript **scripts** versus JavaScript **modules**.
- Details of how a JavaScript program is recognized as a script or module depends on the JavaScript environment.
- Within JavaScript modules, `import` and `export` statements are recognized.
- Within JavaScript scripts, `import` and `export` statements are not recognized and will cause a syntax error.
- There can only be a single ES6 module per file and a ES6 module is restricted to being defined within a single file.
- The code within an ES6 module is always `strict`, as though there was a `"use strict";` declaration at the start of the file.

Exporting Symbols from an ES6 Module

- All definitions within an ES6 module are private to that module unless explicitly `export`'ed.
- Can `export` each definition as it is made:

```
export const CONST = ...
export class Class { ... }
export function fn(...) { ... };
```

- Alternately, can export a list of symbols:

```
const CONST = ...
class Class { ... };
function fn(...) { ... };

export { CONST, Class, fn };
```

Importing Symbols from an ES6 Module

- Can import features from an external module using an `import` statement:

```
import { CONST, Class, fn }
  from './modules/module.js';
...
... CONST + 2 ...
... new Class() ...
... fn() ...
```

- Prefer to use a relative rather than absolute path to make it easier to move stuff around.

Importing Entire Module as an Object

Can import entire module as an object:

```
import * as Module from './modules/module.js';
...
... Module.CONST + 2 ...
... new Module.Class() ...
... Module.fn() ...
```

Default Exports

- Can have a single default `export` per module:

```
export default class { //anonymous class
}
```

- Import it giving it a name:

```
import ModuleClass from './modules/module.js';

... new ModuleClass() ...
```

Renaming

- Can use renaming to avoid naming conflicts:

```
const CONST = ...;
import { CONST as MODULE_CONST, Class, fn }
  from './modules/module.js';
```

- Can use similar syntax for renaming in `export` statements.

ES6 Modules Pragmatics

- Modules can use extension `.mjs`, but many tools do not currently recognize that extension, so `.js` still commonly used.
- On server-side, nodejs recognizes `*.mjs` files as modules, but can also recognize `*.js` files as modules provided there is a `"type": "module"` declaration at the top-level within `package.json`.
- Within browser, modules can be pointed to using `<script type="module">`.

Dynamic Module Loading

- Dynamic imports possible by using asynchronous `import()` function.
- Dynamic imports make it possible to import a module determined dynamically or conditionally.

```
> Path = 1
1
> (async function() {
    Path = await import('path');
  })()
Promise { <pending> }
> Path
[Module] {
  _makeLong: [Function: toNamespacedPath],
  ....
}
>
```

Semantic Versioning

Semantic Versioning attempts to avoid *dependency hell*. It uses a 3 part version number: *M.m.r* where each part is a integer without leading zeros.

Revision Number *r* Incremented for bug fixes.

Minor Version *m* Incremented for added functionality which is backward compatible.

Major Version *M* Incremented for incompatible changes which are not backward compatible.

References

- *MDN JavaScript Modules*.
- *ES6 in Depth: Modules* by Jason Orendorff.
- Modules chapter in *Exploring ES6* by Dr, Axel Rauschmayer.