Chapter 57: Modules

Section 57.1: Defining a module

In ECMAScript 6, when using the module syntax (`import`/`export`), each file becomes its own module with a private namespace. Top-level functions and variables do not pollute the global namespace. To expose functions, classes, and variables for other modules to import, you can use the `export` keyword.

```
// not exported
function somethingPrivate() {
    console.log('TOP SECRET')
}


export const PI = 3.14;

export function doSomething() {
    console.log('Hello from a module!')
}

function doSomethingElse(){
    console.log("Something else")
}

export {doSomethingElse}

export class MyClass {
    test() {}
}
```

Note: ES5 JavaScript files loaded via **`<script>`** tags will remain the same when not using `import`/`export`.

Only the values which are explicitly exported will be available outside of the module. Everything else can be considered private or inaccessible.

Importing this module would yield (assuming the previous code block is in `my-module.js`):

```
import * as myModule from './my-module.js';

myModule.PI;                  // 3.14
myModule.doSomething();       // 'Hello from a module!'
myModule.doSomethingElse();   // 'Something else'
new myModule.MyClass();       // an instance of MyClass
myModule.somethingPrivate(); // This would fail since somethingPrivate was not exported
```

Section 57.2: Default exports

In addition to named imports, you can provide a default export.

```
// circle.js
export const PI = 3.14;
export default function area(radius) {
    return PI * radius * radius;
}
```

You can use a simplified syntax to import the default export.

```
import circleArea from './circle';
console.log(circleArea(4));
```

Note that a *default export* is implicitly equivalent to a named export with the name `default`, and the imported binding (`circleArea` above) is simply an alias. The previous module can be written like

```
import { default as circleArea } from './circle';
console.log(circleArea(4));
```

You can only have one default export per module. The name of the default export can be omitted.

```
// named export: must have a name
export const PI = 3.14;

// default export: name is not required
export default function (radius) {
    return PI * radius * radius;
}
```

Section 57.3: Importing named members from another module

Given that the module from the Defining a Module section exists in the file `test.js`, you can import from that module and use its exported members:

```
import {doSomething, MyClass, PI} from './test'

doSomething()

const mine = new MyClass()
mine.test()

console.log(PI)
```

The `somethingPrivate()` method was not exported from the `test` module, so attempting to import it will fail:

```
import {somethingPrivate} from './test'

somethingPrivate()
```

Section 57.4: Importing an entire module

In addition to importing named members from a module or a module's default export, you can also import all members into a namespace binding.

```
import * as test from './test'

test.doSomething()
```

All exported members are now available on the `test` variable. Non-exported members are not available, just as they are not available with named member imports.

Note: The path to the module `'./test'` is resolved by the *loader* and is not covered by the ECMAScript specification - this could be a string to any resource (a path - relative or absolute - on a filesystem, a URL to a network resource, or any other string identifier).

Section 57.5: Importing named members with aliases

Sometimes you may encounter members that have really long member names, such as `thisIsWayTooLongOfAName()`. In this case, you can import the member and give it a shorter name to use in your current module:

```
import {thisIsWayTooLongOfAName as shortName} from 'module'

shortName()
```

You can import multiple long member names like this:

```
import {thisIsWayTooLongOfAName as shortName, thisIsAnotherLongNameThatShouldNotBeUsed as
otherName} from 'module'

shortName()
console.log(otherName)
```

And finally, you can mix import aliases with the normal member import:

```
import {thisIsWayTooLongOfAName as shortName, PI} from 'module'

shortName()
console.log(PI)
```

Section 57.6: Importing with side effects

Sometimes you have a module that you only want to import so its top-level code gets run. This is useful for polyfills, other globals, or configuration that only runs once when your module is imported.

Given a file named `test.js`:

```
console.log('Initializing...')
```

You can use it like this:

```
import './test'
```

This example will print `Initializing...` to the console.

Section 57.7: Exporting multiple named members

```
const namedMember1 = ...
const namedMember2 = ...
const namedMember3 = ...

export { namedMember1, namedMember2, namedMember3 }
```