Chapter 73: Modularization Techniques

Section 73.1: ES6 Modules

Version ≥ 6

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.

Note: Although this is the official method for creating JavaScript modules, it is not supported by any major browsers right now. However, ES6 Modules are supported by many transpilers.

```
export function greet(name) {
    console.log("Hello %s!", name);
}

var myMethod = function(param) {
    return "Here's what you said: " + param;
};

export {myMethod}

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

Using Modules

Importing modules is as simple as specifying their path:

```
import greet from "mymodule.js";

greet("Bob");
```

This imports only the `myMethod` method from our `mymodule.js` file.

It's also possible to import all methods from a module:

```
import * as myModule from "mymodule.js";

myModule.greet("Alice");
```

You can also import methods under a new name:

```
import { greet as A, myMethod as B } from "mymodule.js";
```

More information on ES6 Modules can be found in the Modules topic.

Section 73.2: Universal Module Definition (UMD)

The UMD (Universal Module Definition) pattern is used when our module needs to be imported by a number of different module loaders (e.g. AMD, CommonJS).

The pattern itself consists of two parts:

1. An IIFE (Immediately-Invoked Function Expression) that checks for the module loader that is being implemented by the user. This will take two arguments; `root` (a `this` reference to the global scope) and `factory` (the function where we declare our module).
2. An anonymous function that creates our module. This is passed as the second argument to the IIFE portion of the pattern. This function is passed any number of arguments to specify the dependencies of the module.

In the below example we check for AMD, then CommonJS. If neither of those loaders are in use we fall back to making the module and its dependencies available globally.

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['exports', 'b'], factory);
    } else if (typeof exports === 'object' && typeof exports.nodeName !== 'string') {
        // CommonJS
        factory(exports, require('b'));
    } else {
        // Browser globals
        factory((root.commonJsStrict = {}), root.b);
    }
}(this, function (exports, b) {
    //use b in some fashion.

    // attach properties to the exports object to define
    // the exported module properties.
    exports.action = function () {};
}));
```

Section 73.3: Immediately invoked function expressions (IIFE)

Immediately invoked function expressions can be used to create a private scope while producing a public API.

```
var Module = (function() {
  var privateData = 1;

  return {
    getPrivateData: function() {
      return privateData;
    }
  };
})();
Module.getPrivateData(); // 1
Module.privateData; // undefined
```

See the Module Pattern for more details.

Section 73.4: Asynchronous Module Definition (AMD)

AMD is a module definition system that attempts to address some of the common issues with other systems like CommonJS and anonymous closures.

AMD addresses these issues by:

- Registering the factory function by calling define(), instead of immediately executing it
- Passing dependencies as an array of module names, which are then loaded, instead of using globals
- Only executing the factory function once all the dependencies have been loaded and executed

- Passing the dependent modules as arguments to the factory function

The key thing here is that a module can have a dependency and not hold everything up while waiting for it to load, without the developer having to write complicated code.

Here's an example of AMD:

```
// Define a module "myModule" with two dependencies, jQuery and Lodash
define("myModule", ["jquery", "lodash"], function($, _) {
    // This publicly accessible object is our module
    // Here we use an object, but it can be of any type
    var myModule = {};

    var privateVar = "Nothing outside of this module can see me";

    var privateFn = function(param) {
        return "Here's what you said: " + param;
    };

    myModule.version = 1;

    myModule.moduleMethod = function() {
        // We can still access global variables from here, but it's better
        // if we use the passed ones
        return privateFn(windowTitle);
    };

    return myModule;
});
```

Modules can also skip the name and be anonymous. When that's done, they're usually loaded by file name.

```
define(["jquery", "lodash"], function($, _) { /* factory */ });
```

They can also skip dependencies:

```
define(function() { /* factory */ });
```

Some AMD loaders support defining modules as plain objects:

```
define("myModule", { version: 1, value: "sample string" });
```

Section 73.5: CommonJS - Node.js

CommonJS is a popular modularization pattern that's used in Node.js.

The CommonJS system is centered around a `require()` function that loads other modules and an `exports` property that lets modules export publicly accessible methods.

Here's an example of CommonJS, we'll load Lodash and Node.js' `fs` module:

```
// Load fs and lodash, we can use them anywhere inside the module
var fs = require("fs"),
    _ = require("lodash");

var myPrivateFn = function(param) {
    return "Here's what you said: " + param;
};
```

```
// Here we export a public `myMethod` that other modules can use
exports.myMethod = function(param) {
    return myPrivateFn(param);
};
```

You can also export a function as the entire module using `module.exports`:

```
module.exports = function() {
    return "Hello!";
};
```