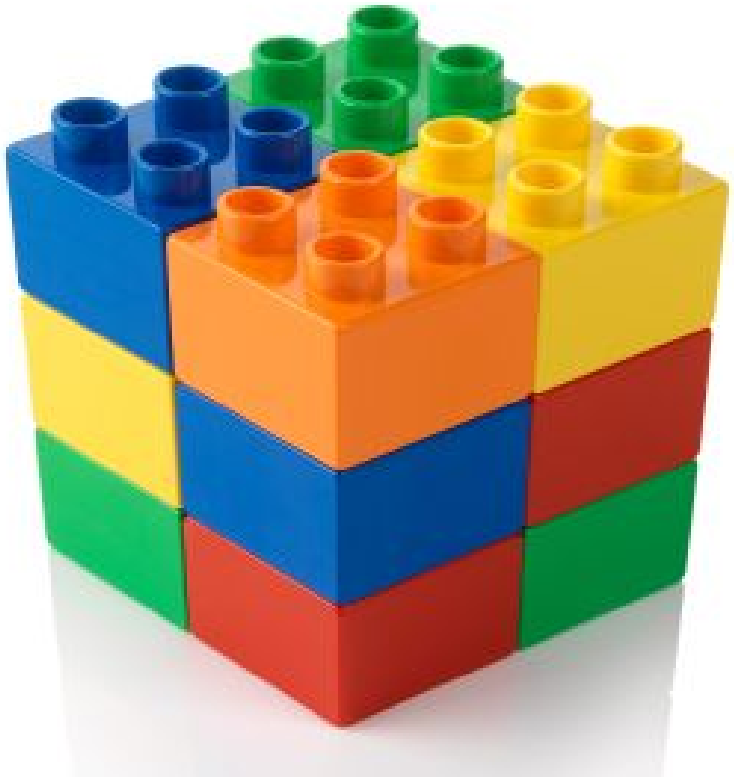
Modules in JavaScript



Why Modules?

- Larger program need a larger unit of organization than individual functions
- And due to the lack of namespaces, JavaScript needs a mechanism that can minimize the pollution of the global namespace

- **Modules** divide programs into clusters of code and helps to structure large programs and minimize the pollution of the global namespace

Using Functions as Namespaces

- Functions are the only things in JavaScript that create a new scope

```
var names1 = ["Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"];
function dayName1(number) {
    return names1[number];
}

console.log(dayName1(1));
```

```
var dayName2 = function () {
    var names2 = ["Sunday", "Monday", "Tuesday", "Wednesday",
                  "Thursday", "Friday", "Saturday"];
    return function (number) {
        return names2[number];
    };
}();

console.log(dayName2(3));
```

Objects as Export Interface

- When we want to export more than one function we can use an object to hold all the functions and variables we want to export

```
(function (exports) {
    var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday"];

    exports.name = function (number) {
        return names[number];
    };
    exports.number = function (name) {
        return names.indexOf(name);
    };
})(this.weekDay = {});

console.log(weekDay.name(5));
console.log(weekDay.number("Saturday"));
```

Detaching from the global scope

- We can create a system that allows one module to directly ask for the interface object of another module, without going through the global scope

A minimal implementation of require

```
function require(name) {
    var code = new Function("exports", getFile(name));
    var exports = {};
    code(exports);
    return exports;
}
```

weekDay.js

```
var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
             "Thursday", "Friday", "Saturday"];
exports.name = function (number) {
    return names[number];
};
exports.number = function (name) {
    return names.indexOf(name);
};
```

```
var weekday = require("weekDay");
console.log(weekday.name(3));
```

An Enhanced require

```
function require(name) {
    if (name in require.cache)
        return require.cache[name];

    var code = new Function("exports, module", readFile(name));
    var exports = {}, module = { exports: exports };
    code(exports, module);

    require.cache[name] = module.exports;
    return module.exports;
}
require.cache = Object.create(null);
```

- This style of module system is called **CommonJS modules**
  - It is built into the Node.js system

CommonJS In The Browser

- Reading a file (module) from the Web is a lot slower than reading it from the hard disk
- While a script is running in the browser, nothing else can happen to the website on which it runs

- To overcome the problem with slow loading of modules in the browser with CommonJS you can run Browserify on your code before you serve it on a web page
  - http://browserify.org/

The Asynchronous Module Definition (AMD)

- Another solution to the slow loading of modules is to load them asynchronously in the background and then call the function, initializing the module, when the dependencies have been loaded
- This is what the AMD version of require does

- The RequireJS project (requirejs.org) provides a popular implementation of this style of module loader

Require.js (AMD) Example

How the scripts are included inside **the HTML file**:

```
<script data-main="scripts/main" src="scripts/require.js"></script>
```

main.js is used for initialization:

```
require(["purchase"], function (purchase) {
    purchase.purchaseProduct();
});
```

- In RequireJS, all code is wrapped in **require**() or **define**() functions
- The first parameter to these functions specifies dependencies
- The second parameter is an anonymous function which takes an object that is used to call the functions inside the dependent file

Multiple dependencies can be loaded using the following syntax:

```
require(["a","b","c"], function(a,b,c){
  …
});
```

Use define to export from module

purchase.js module:

```
define(["credits", "products"], function (credits, products) {

    console.log("Function : purchaseProduct");

    return {
        purchaseProduct: function () {

            var credit = credits.getCredits();
            if (credit > 0) {
                products.reserveProduct();
                return true;
            }
            return false;
        }
    }
});
```

Independent Modules

- products.js and credits.js are not dependent on anything

products.js module:

```
define(function () {
    return {
        reserveProduct: function () {
            console.log("Function : reserveProduct");

            return true;
        }
    }
});
```

credits.js module:

```
define(function () {
    console.log("Function : getCredits");

    return {
        getCredits: function () {
            var credits = "100";
            return credits;
        }
    }
});
```

require() vs. define()

- The require()function is used to run immediate functionalities
  - To import one or more modules
- The define() function is used to define modules for use in multiple locations
  - To export functionality and optionally import other functionality

Modules in ES2015

- In ES2015 modules is supported in JavaScript
- ES6 modules are stored in files
  - There is exactly one module per file and one file per module
- In browsers: scripts versus modules

	Scripts	Modules
HTML element	<script>	<script type="module">
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of `this` at top level	window	undefined
Executed	synchronously	asynchronously
Has `import` statement	No	Yes
Programmatic imports	Yes	Yes
File extension	.js	.js

This feature is not implemented in any browsers natively at this time! But it is implemented in many transpilers, such as the Traceur Compiler, Babel, TypeScript or Webpack.

Multiple named exports

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
}
```

You can specify the items you want to import

```
//------ main.js ------
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

Or you can import the complete module

```
//------ myModule.js ------
import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

Single default export

- There can be a single default export
  - can be a function, a class, an object or anything else

```
//------ lib1.js ------
export default {
  field1: value1,
  field2: value2
};
```

```
//------ main2.js ------
import lib1 from 'lib1';
Console.log(lib1.field1);
```

```
//------ lib2.js ------
export default function () {
 ...
}
```

```
//------ main2.js ------
import myFunc from 'lib2';
myFunc();
```

```
//------ lib3.js ------
export default class {
 ...
}
```

```
//------ main3.js ------
import MyClass from 'lib3';
const inst = new MyClass();
```

References & Links

- Eloquent JavaScript
  http://eloquentjavascript.net/10_modules.html
- CommonJS
  http://requirejs.org/docs/commonjs.html
- Browserify
  http://browserify.org/
- RequireJS
  http://requirejs.org
  http://www.sitepoint.com/understanding-requirejs-for-effective-javascript-module-loading/
- ES2015 Modules
  https://hacks.mozilla.org/2015/08/es6-in-depth-modules/
  http://exploringjs.com/es6/ch_modules.html