Lecture 2.0

Modules

Modules

Modules

Good authors divide their books into chapters and sections; good programmers divide their programs into modules.

Like a book chapter, modules are just clusters of words (or code, as the case may be).

Good modules, however, are highly self-contained with distinct functionality, allowing them to be shuffled, removed, or added as necessary, without disrupting the system as a whole.

Why use modules?

1. Maintainability

By definition, a module is self-contained. A well-designed module aims to lessen the dependencies on parts of the codebase as much as possible, so that it can grow and improve independently.

Updating a single module is much easier when the module is decoupled from other pieces of code.

(Going back to our book example, if you wanted to update a chapter in your book, it would be a nightmare if a small change to one chapter required you to tweak every other chapter as well. Instead, you'd want to write each chapter in such a way that improvements could be made without affecting other chapters.)

Why use modules?

2. Namespacing

In JavaScript, variables outside the scope of a top-level function are global (meaning, everyone can access them). Because of this, it's common to have "namespace pollution", where completely unrelated code shares global variables.

Sharing global variables between unrelated code is a big no-no in development.

Modules allow us to avoid namespace pollution by creating a private space for our variables.

Why use modules?

3. Reusability

We've all copied code we previously wrote into new projects at one point or another. For example, let's imagine you copied some utility methods you wrote from a previous project to your current project.

That's all well and good, but if you find a better way to write some part of that code you'd have to go back and remember to update it everywhere else you wrote it.

This is obviously a huge waste of time. Wouldn't it be much easier if there was code we could reuse again and again?

What is a module?

A module is just a file. One script is one module. As simple as that.

Modules can load each other and use special directives export and import to interchange functionality, call functions of one module from another one:

- export keyword labels variables and functions that should be accessible from outside the current module.
- import allows the import of functionality from other modules.

- Node.js equivalent is exports and require

CommonJs

CommonJS

- CommonJS is a volunteer working group that designs and implements JavaScript APIs for declaring modules.

- A CommonJS module is essentially a reusable piece of JavaScript which exports specific objects, making them available for other modules to *require* in their programs. If you've programmed in Node.js, you'll be very familiar with this format.

CommonJs Modules

With CommonJS, each JavaScript file stores modules in its own unique module context (just like wrapping it in a closure).

In this scope, we use the *module.exports* object to expose modules, and *require* to import them.

CommonJS Module Example

```
function greeterModule() {

    this.hello = function() {
      console.log('hello!');
    }

    this.goodbye = function() {
      console.log('goodbye!');
    }
  }

  module.exports = greeterModule;
```

CommonJS Module Example

- We use the special object module and place a reference of our function into *module.exports*. This lets the CommonJS module system know what we want to expose so that other files can consume it.

```
var greetModule = require('./greeterModule');

var greeterInstance = new greetModule();
greeterInstance.hello(); // 'hello!'
greeterInstance.goodbye(); // 'goodbye!
```