JAVASCRIPT MODULE PATTERNS

nic jansma | nicj.net | @nicj

JAVASCRIPT OBJECTS

What is a JavaScript object?

{ }

`{}:`

- A collection of properties
- Each property has a value
- A value can be a `number`, `string`, `boolean`, `object` or `function`

WHAT ISN'T AN OBJECT

Only `null` and `undefined` are not objects

HOW DO YOU CREATE OBJECTS? VERSION 1

Using an object initializer {}:

```
// create an empty object
var emptyObject = {};

// create an object with properties
var obj = {
    stringProperty: "hello",
    integerProperty: 123,
    functionProperty: function() { return 0; },
    "a property with spaces": false,
    subObject: {
        booleanProperty: true
    }
};
```

HOW DO YOU CREATE OBJECTS? VERSION 2

Using a constructor `function` (new keyword):

```
// create an empty object
var emptyObject = new Object();

// define an object constructor
function Keg(contains, amount) {
    this.contains = contains;
    this.amount   = amount;
}

// create an object
var keg = new Keg("Soda", 100.0);
```

HOW DO YOU CREATE OBJECTS? VERSION 3

Using `Object.create()`:

```
// create an empty object
var emptyObject = Object.create(Object.prototype);

// define an object with default properties
var Keg = {
    contains: "Unknown",
    amount:   0.0
}

// create an object
var keg      = Object.create(Keg);

// modify its properties
keg.contains = "Soda";
keg.abv      = 100.0;
```

JAVASCRIPT MODULE PATTERNS

- A **module** helps keep units of code cleanly separated and organized
- A **pattern** is a common technique that can be re-used and applied to every-day software design problems
- **JavaScript Module Patterns** help us organize and limit code scope in any project

JAVASCRIPT MODULES

- The JavaScript language doesn't have **classes**, but we can emulate what classes can do with **modules**
- A module helps **encapsulate** data and functions into a single component
- A module limits **scope** so the variables you create in the module only live within it
- A module gives **privacy** by only allowing access to data and functions that the module wants to expose

BASIC OBJECT

Let's build a module for a Keg that can be filled with soda. It has two basic properties:

```
function Keg(contains, amount) {
    this.contains = contains;
    this.amount   = amount;
}
```

BASIC OBJECT

We can add a `fill()` function so others can fill it with something tasty:

```
function Keg(contains, amount) {
    this.contains = contains;
    this.amount   = amount;
    this.fill     = function(beverage, amountAdded) {
        this.contains = beverage;
        this.amount = amountAdded;
    };
}
```

BASIC OBJECT

Right now, all of the Keg's properties are public. The world has full access to change our data:

```
var keg = new Keg();
keg.fill("Soda", 100.0);
keg.amount = 9999; // oh no! they accessed our internal data
```

BASIC MODULE PATTERN: CONSTRUCTORS

Let's switch to the Module Pattern, which gives us the ability to have public and private members:

```
// define the constructor
function Keg(_contains, _amount) {
    // private members
    var contains = _contains;
    var amount   = _amount;

    // public methods
    return {
        fill: function(beverage, amountAdded) {
            contains = beverage;
            amount = amountAdded;
        }
    }
}

// create an instance of a Keg
var keg = new Keg("Soda", 100.0);

// modify its properties
keg.fill("Pop", 50.0); // this is the only public member
var amt = keg.amount;  // undefined! hidden from us
```

BASIC MODULE PATTERN: CONSTRUCTORS

We can add additional methods to give access to our private variables without changing them:

```
function Keg(_contains, _amount) {
    /* ... private members ... */
    return {
        fill: function() { ... },
        getAmount: function() {
            return amount;
        },
        getContents: function() {
            return contains;
        }
    }
}

var keg = new Keg("Soda", 100.0);
var amt = keg.getAmount(); // 100.0

keg.fill("Pop", 50.0);
amt = keg.getAmount(); // 50.0
```

BASIC MODULE PATTERN: CONSTRUCTORS

You can have private functions as well:

```
function Keg(_contains, _amount) {
    // private members
    var contains = _contains;
    var amount   = _amount;

    // private function
    function updateAmount(newAmount) {
        if (newAmount < 0) { newAmount = 0; }
        amount = newAmount;
    }

    // public methods
    return {
        fill: function(beverage, amountAdded) {
            contains = beverage;
            updateAmount(amountAdded);
        }
        /* ... */
    }
}
```

BASIC MODULE PATTERN: CONSTRUCTORS

Completed:

```
function Keg(_contains, _amount) {
    // private members
    var contains = _contains;
    var amount   = _amount;

    // private function
    function updateAmount(newAmount) {
        if (newAmount < 0) { newAmount = 0; }
        amount = newAmount;
    }

    // public methods
    return {
        fill: function(beverage, amountAdded) {
            contains = beverage;
            updateAmount(amountAdded);
        },
        getAmount: function() {
            return amount;
        },
        getContents: function() {
            return contains;
        }
    }
```

DISADVANTAGES

- The Basic Module Pattern for constructing objects has one big disadvantage: you're not taking advantage of **prototypes**
- A prototype is a value (`number`, `string`, `function`, etc) that you can assign to *all* instances of a class using `ClassName.prototype.`
- Instead of each instance having a *copy* of the member, the single prototype member is shared
- This gives you substantial memory savings if you have many instances of the object

KEG USING PROTOTYPE

Instead of each instance having it's own version of the same `fill()` function, there's one global `Keg.prototype.fill`:

```
function Keg(contains, amount) {
    // these now need to be public members
    this.contains = contains;
    this.amount   = amount;
}

Keg.prototype.fill = function(beverage, amountAdded) {
    // because this doesn't have access to 'vars' in the Keg function
    this.contains = beverage;
    this.amount = amountAdded;
};
Keg.prototype.getAmount = function() {
    return this.amount;
};
Keg.prototype.getContents = function() {
    return this.contains;
};
```

KEG USING PROTOTYPE

- The Keg's internal properties (`contains` and `amount`) need to change from being defined within the Keg function's closure (`var contains = ...`) to be public properties (`this.contains = ...`)
- This is because the `Keg.prototype.fill` function wasn't defined within the Keg's function closure, so it would have no visibility to `var`s defined within it
- Thus the properties can be modified by anyone, outside of the protection of your module

BASIC MODULE PATTERN: NON-CONSTRUCTORS

- If your module is a "global object" instead of a constructor (i.e. `jQuery`), you can simplify the syntax a bit
- Wrap it up in an immediately-invoked functional expression (IIFE) to get closure for your private variables

BASIC MODULE PATTERN: NON-CONSTRUCTORS

```
var KegManager = (function() {
    var kegs = [];

    // exports
    return {
        addKeg: function(keg) { kegs.push(keg); }
        getKegs: function() { return kegs; }
    }
})();

var sodaKeg = new Keg("Soda", 100.0);
KegManager.addKeg(sodaKeg);

var kegs = KegManager.getKegs(); // a list of Keg objects
```

IMPORTS

If you want to "import" other global variables or other modules, they can be passed in as IIFE arguments:

```
var KegManager = (function($) {
    var kegs = [];

    // do something with $

    // exports
    return {
        addKeg: function(keg) { kegs.push(keg); }
        getKegs: function() { return kegs; }
    }
})(jQuery);

var sodaKeg = new Keg("Soda", 100.0);
KegManager.addKeg(sodaKeg);

var kegs = KegManager.getKegs(); // a list of Keg objects
```

REVEALING MODULE PATTERN

- An update to the Module Pattern
- Define everything first, then return an object that has properties for the items you want to export (make public)

REVEALING MODULE PATTERN

```
function Keg(_contains, _amount) {
    // private members
    var contains = _contains;
    var amount   = _amount;

    // private functions
    function updateAmount(newAmount) {
        if (newAmount < 0) { newAmount = 0; }
        amount = newAmount;
    }

    // public functions
    function fill(beverage, amountAdded) {
        contains = beverage;
        updateAmount(amountAdded);
    }

    function getAmount() { return amount; }
    function getContents() { return contains; }

    // exports
    return {
        fill: fill,
        getAmount: getAmount,
        getContents: getContents
    }
}
```

REVEALING MODULE PATTERN

Pros:

- All public and private members are defined in the same way
- All exports are listed in an easy-to-read list at the end
- If someone were to "patch" (overwrite) an export, your internal functions still refer to your own implementation

COMMONJS

- A module standard
- Commonly used on the server (NodeJS)
- Each file is a (single) module, each module is a (separate) file
- A global `exports` variable is available that you can assign your exports to

COMMONJS MODULE DEFINITION

A file contains a single module:

keg.js

```
// imports
var KegManager = require("kegmanager");

// constructor we'll export
function Keg(_contains, _amount) {
    // ... same as before

    // tell the KegManager about this new keg
    KegManager.add(this);
}

// some other private vars
var foo = false;

// exports
exports.Keg = Keg;
```

COMMONJS MODULE USAGE

Same as module definition:

```
var Keg = require("./keg").Keg;
var keg = new Keg("Soda", 100);
```

AMD

- Asynchronous Module Definition
- Commonly used in the browser (Dojo, MooTools, jQuery)
- Allows for modules and their dependencies to be loaded asynchronously
- Need to use a "loader", such as RequireJS (http://requirejs.org/)

AMD MODULE DEFINITION: `DEFINE`

Defines a module, its dependencies, and the initialization function that runs once all dependencies are loaded:

```
define(
    "Keg",                  // module name, optional but suggested
    ["KegManager"],         // list of dependencies
    function(KegManager) {  // initialization function
        // constructor we'll export
        function Keg(_contains, _amount) {
            // ... same as before

            // tell the KegManager about this new keg
            KegManager.add(this);
        }

        // some other private vars
        var foo = false;

        // exports
        return {
            Keg: Keg
        }
});
```

AMD MODULE USAGE: `REQUIRE`

Load the modules you need

```
require(
    ["Keg"],
    function(Keg) {
        // will only run once Keg (and its dependency, KegManager) is loaded
        var keg = new Keg.Keg("Soda", 100);
});
```

REQUIREJS

- AMD specifies a **format** for how to define a module and its dependencies
- It's up to a **loader** to figure out how to fetch and run the modules in the correct load order
- RequireJS (and its little sister almond) are the best loader options today

REQUIREJS USAGE

```
<!DOCTYPE html>
<html>
    <head>
        <title>My Sample Project</title>
        <!-- data-main attribute tells require.js to load
             scripts/main.js after require.js loads. -->
        <script data-main="scripts/main" src="scripts/require.js"></script>
    </head>
    <body>
        <h1>My Sample Project</h1>
    </body>
</html>
```

scripts/main.js

```
require(['app/module1', 'app/module2']);
```

REQUIREJS OPTIMIZER

Builds all of your modules into a single file (great for deployment)

Install requirejs:

```
> npm install -g requirejs
```

Optimize your JavaScript:

```
> node r.js -o baseUrl=. name=main out=main-built.js
```

ALMOND

https://github.com/jrburke/almond

- A replacement AMD loader for RequireJS
- Minimal AMD loader API footprint
- Only used for bundled AMD modules (not dynamic loading)

UMD

- Universal Module Definition
- Code templates for defining a module that works in multiple environments, such as AMD, CommonJS and the browser
- https://github.com/umdjs/umd

UMD

Defines a module that works in Node, AMD and browser globals

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['b'], factory);
    } else if (typeof exports === 'object') {
        // Node. Does not work with strict CommonJS, but
        // only CommonJS-like environments that support module.exports,
        // like Node.
        module.exports = factory(require('b'));
    } else {
        // Browser globals (root is window)
        root.returnExports = factory(root.b);
    }
}(this, function (b) {
    //use b in some fashion.

    // Just return a value to define the module export.
    // This example returns an object, but the module
    // can return a function as the exported value.
    return {};
}));
```

https://github.com/umdjs/umd/blob/master/returnExports.js

THE FUTURE: ES6 MODULES

Goals:

- Compact syntax (similar to CommonJS)
- Support for asynchronous loading and configurable module loading (similar to AMD)

ES6 MODULES

keg.js

```
module Keg {
    // imports
    import { KegManager} from 'kegmanager';

    // constructor we'll export
    export function Keg(_contains, _amount) {
        // ... same as before

        // tell the KegManager about this new keg
        KegManager.add(this);
    }
}
```

FURTHER READING

- JavaScript Design Patterns - Addy Osmani: http://addyosmani.com/resources/essentialjsdesignpatterns/bo
- Writing Modular JavaScript With AMD, CommonJS & ES Harmor Addy Osmani: http://addyosmani.com/writing-modular-js/
- ECMAScript 6 modules: the final syntax - Axel Rauschmayer: http://www.2ality.com/2014/09/es6-modules-final.html

THNX