JavaScript Modules Patterns

Onkar Deshpande

XORIANT

JavaScript Objects

XORIANT

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?

Using an object initializer {}:

Version 1

```
// 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?

Using a constructor function (new keyword):

Version 2

```
// 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?

Using Object.create():

Version 3

```
// 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

JavaScript Module Patterns

- A **module** helps keep units of code cleanly separated & organized
- A **pattern** is a common technique that can be re-used & 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 vars 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

- 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

Thank You!

Onkar Deshpande
Senior Trainer

XORIANT

Mobile: +91 9930777883
deshpande.onkar@xoriant.com
www.xoriant.com

US – Corporate Headquarters

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

20 Broadwick Street
Soho, London
W1F 8HT, UK

89 Worship Street
Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

India

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076

Pune
5th Floor, Amar Paradigm
Baner, Pune 411 045

Kolkata
2B, 12th Floor, Tower 'C'
Rajarhat, Kolkata 700 156

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road,
Koramangala 1st Block,
Bengaluru (Bangalore) 560034

Gurgaon
A/373rd Floor, Sigma Center
Gurgaon, Haryana 122 011s