JAVASCRIPT PT I

The basics of JS and how to write good JS from the start.

CLARA WU

@scwu

@sclarawu

s.clara.wu

sclarawu@gmail.com

clarawu.com

Junior in NETS, WiCS board, PennApps Labs
(you should apply!)

WHAT IS JS

- generally has been used client side (but with Node.JS, has shifted to be full-stack)

- dynamically typed - with asynchronous property (not synchronized, can run function and run another function without a return from the previous function

  - single-threaded-> so asynchronous becomes necessary

- often used to manipulate HTML pages in some way, and can be injected into any website w/o installing anything (interpreted by the browser, why it's called "client-side")

WHERE DOES JQUERY FIT IN?

- You can use JS for anything - algorithms, data structures, etc but it's most typical use is for web programming.
    - There are many different browsers that interpret JS slightly differently.
    - jQuery was built specifically for web programming - and is a library that gives you a cross-browser set of functions to use syntactually different from JS, but functionally similar that will be interpreted the same across all browsers.
    - jQuery almost generally makes everything easier.

PROS/CONS OF JS

- Pros
  - Easy to debug (Chrome Developer Tools -> you can print directly to the console)
  - Is interpreted on the browser. Easy and can lighten load on server.
- Cons
  - Privacy issues -> your code can be seen by the person viewing your site. (no passwords, secret keys, etc)
  - single threaded
  - scoping issues

WHAT IS A JS OBJECT?

- JS is built off of what we call a JavaScript object.
- Almost everything in JS is an object (booleans, numbers, strings, arrays)
- An object in JS is a collection of properties that you can make. These associations of properties are similar to a map.

```
var clara = {};
clara.name = "Clara Wu";
clara['birthday'] = 09161993;

function Person(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

var clara = new Person("Clara Wu", 09161993);
console.log(clara.name); //would print out Clara Wu to the log
```

ASPECTS TO GOOD JS

- People forget that JS is still a programming language. If you wouldn't write all of your Java code in one main method - why would you do it in Javascript?

- Sometimes it's not totally clear how to do this for the web, however.

- **Namespacing**

- **Object Oriented Programming**

- **Modularization**

NAMESPACING

- Global variables are bad - you should avoid populating the global namespace because it gets messy very fast.
  - Imagine setting "var count =0;" globally. What happens if you forget you set this globally and you've used count twice? This does not lead to manageable code.
- Instead, for every feature you can create an object and then for every function within that - we can make it a property of that object. The variables within those properties will be local.

NAMESPACING CONT.

```
// Create a namespace / module for your project
window.MyModule = {};

// Commence scope to prevent littering
// the window object with unwanted variables
(function() {

    var Animal = window.MyModule.Animal = Object.extend(Object, {
         move: function() {alert('moving...');}
    });

        // .. more code

})();
```

Stick to namespaces so the code above should be run within its own context outside the window object, this is critical if you intend your code to run as one of many concurrent frameworks / libraries executing in the browser window.

OBJECT ORIENTED PROGRAMMING

- prototype-based - you have a function constructor
- When you have methods of that object - you make them prototypes of that object.
- Similar to classes in Java but not quite. Same idea.

```
function Car( model, year, miles ) {

  this.model = model;
  this.year = year;
  this.miles = miles;

  this.toString = function () {
    return this.model + " has done " + this.miles + " miles";
  };
}

// Usage:

// We can create new instances of the car
var civic = new Car( "Honda Civic", 2009, 20000 );
var mondeo = new Car( "Ford Mondeo", 2010, 5000 );

// and then open our browser console to view the
// output of the toString() method being called on
// these objects
console.log( civic.toString() );
console.log( mondeo.toString() );
```

MODULARIZATION

- Exactly what is sounds like - breaking your web-app into different modules of interactivity (whether this means physical blocks or features depends)

- This mimics encapsulation that we know about.

- Makes code more maintanable.

- Use of object literals - which are represented as {}, similar idea to name-spacing but for different reasons.

```
var myNamespace = (function () {

  var myPrivateVar, myPrivateMethod;

  // A private counter variable
  myPrivateVar = 0;

  // A private function which logs any arguments
  myPrivateMethod = function( foo ) {
      console.log( foo );
  };

  return {

    // A public variable
    myPublicVar: "foo",

    // A public function utilizing privates
    myPublicFunction: function( bar ) {

      // Increment our private counter
      myPrivateVar++;

      // Call our private method using bar
      myPrivateMethod( bar );

    }
  };

})();
```

MODULARIZATION CONT

```
var basketModule = (function () {
  // privates
  var basket = [];

  function doSomethingPrivate() {
    //...
  }

  function doSomethingElsePrivate() {
    //...
  }

  // Return an object exposed to the public
  return {
    // Add items to our basket
    addItem: function( values ) {
      basket.push(values);
    },

    // Get the count of items in the basket
    getItemCount: function () {
      return basket.length;
    },

    // Public alias to a  private function
    doSomething: doSomethingPrivate,

    // Get the total value of items in the basket
    getTotal: function () {

      var q = this.getItemCount(),
          p = 0;

      while (q--) {
        p += basket[q].price;
      }

      return p;
    }
  };
}());
```

```
// basketModule returns an object with a public API we
can use

basketModule.addItem({
  item: "bread",
  price: 0.5
});

basketModule.addItem({
  item: "butter",
  price: 0.3
});

// Outputs: 2
console.log( basketModule.getItemCount() );

// Outputs: 0.8
console.log( basketModule.getTotal() );

// However, the following will not work:

// Outputs: undefined
// This is because the basket itself is not exposed as
a part of our
// the public API
console.log( basketModule.basket );

// This also won't work as it only exists within the
scope of our
// basketModule closure, but not the returned public
object
console.log( basket );
```

OTHER DESIGN PATTERNS

- Like the Singleton pattern which is very similar, but restricts instantiation of a class to a single object.

```
var mySingleton = (function () {

  // Instance stores a reference to the Singleton
  var instance;

  function init() {

    // Singleton

    // Private methods and variables
    function privateMethod(){
        console.log( "I am private" );
    }

    var privateVariable = "Im also private";

    var privateRandomNumber = Math.random();

    return {

      // Public methods and variables
      publicMethod: function () {
        console.log( "The public can see me!" );
      },

      publicProperty: "I am also public",

      getRandomNumber: function() {
        return privateRandomNumber;
      }

    };

  };
  return {

    // Get the Singleton instance if one exists
    // or create one if it doesn't
    getInstance: function () {

      if ( !instance ) {
        instance = init();
      }

      return instance;
    }

  };

})();
```

```
// Usage:

var singleA = mySingleton.getInstance();
var singleB = mySingleton.getInstance();
console.log( singleA.getRandomNumber() ===
    singleB.getRandomNumber() ); // true

var badSingleA = myBadSingleton.getInstance();
var badSingleB = myBadSingleton.getInstance();
console.log( badSingleA.getRandomNumber() !==
    badSingleB.getRandomNumber() ); // true
```

OTHER USEFUL THINGS TO KNOW

- Javascript-heavy apps will use encapsulation libraries (my favorite and the most popular is Require.JS)

- It'll manage loading modules in the correct order and controls which scripts are loaded where.

- Is essentially necessary to make your JS-heavy app managable.

```
require({

    paths: {
        "jquery": "https://ajax.googleapis.com/ajax/libs/
jquery/1.7.1/jquery.min",
        "jqueryui": "https://ajax.googleapis.com/ajax/libs/
jqueryui/1.8.18/jquery-ui.min",
        "boilerplate": "../patterns/jquery.widget-
factory.requirejs.boilerplate"
    }
}, ["require", "jquery", "jqueryui", "boilerplate"],
function (req, $) {

    $(function () {

        var instance = $("#elem").myWidget();
        instance.myWidget("methodB");

    });
});
```

RESOURCES

- http://addyosmani.com/resources/essentialjsdesignpatterns/book/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/
- http://eloquentjavascript.net/