Javascript Essentials

a Keyhole Software tutorial

This tutorial covers:

- ✔ JavaScript Execution Environment
- ✔ The structure of the JavaScript language
- ✔ The importance of Objects
- ✔ Prototypes and Inheritance
- ✔ Functions and Closures
- ✔ AJAX

If you've been developing enterprise web applications, it's likely that you have applied JavaScript in some fashion - probably to validate user input with a JavaScript function that validates a form control, manipulate an HTML document object model (DOM) for a user interface effect, or even to use AJAX to access the server to eliminate a page refresh.

Single Page Application architectures allow rich, responsive application user interfaces to be developed. There are many frameworks and approaches available, excluding plug-in technologies, that are JavaScript-based. This means that developers need a deeper understanding of the JavaScript language features. This tutorial assumes you have programming experience in a traditional object oriented language like Java or C#, and introduces features of JavaScript that allows it to be a general purpose programming language. You may be surprised by its expressiveness and object oriented capabilities.

Keyhole Software, LLC. 8900 State Line Road, Suite 455 Leawood, KS 66206 Tel 877-521-7769 www.keyholesoftware.com

TABLE OF CONTENTS

1. ENVIRONMENT

One clue that JavaScript was not originally intended to be a general purpose language is the fact that a browser is required to execute it. The snippet below shows how an HTML page loads a JavaScript function defined inline. Normally this assumes the HTML page and JavaScript file reside on a web server.

Listing 1 – HTML page loading a JavaScript function

```
<script>
  function sayhello() {
      alert('hello world');
  }
</script>

...

<input type="button" value="say hello" onclick="sayhello();"/>
```

The `sayhello()` function defined above can invoked and executed in a variety of ways, including:

1. Putting inline JavaScript tags at the beginning or end of the file when the HTML form button is clicked.
2. Calling the function when a form button is clicked.
3. Putting `<script> </script>` elements at the beginning or end of an HTML document, depending on the browser you're using.
4. Executing JavaScript on a page load using the jQuery framework.

As you can see, there is not any kind of main method or entry point mechanism like other languages, so a browser and an HTML page load of some kind is required to execute JavaScript. Some server side Java solutions have recently become available, but generally speaking JavaScript for UI development requires a browser.

Open Source Steps Up

Luckily, innovations of the open source community have filled this need. Environments have been created that allow JavaScript to be executed outside of a browser, commonly referred to as "headless," or server side JavaScript.

Node.js is one popular open source framework that provides a JavaScript runtime environment outside of a browser. With Node.js, JavaScript can be executed from a command line or by specifying files. Node.js is also available for most operating systems. Phantom.js is another viable option on the market. Although similar, the intent of the headless environments is different between the two. Phantom.js has HTML DOM (Document Object Model) available while Node.js does not. But both still provide a way to develop and test code outside of a browser and web server. Here are links to these projects:

- Node – http://nodejs.org/
- Phantom – http://phantomjs.org/

The examples presented in this tutorial can all be typed into and executed with a headless JavaScript environment. Assuming Node.js or Phantom.js binaries have been installed on your operating system, you can execute the previous JavaScript file with the expressions that follow:

```
// Java Script defined
// in HelloWorld.js text file

function helloWorld() {
  console.log("hello world")
}
helloWorld();

// Execute JavaScript
$node helloworld.js
$phantomjs helloworld.js
```

```
// Executing with a node console
```

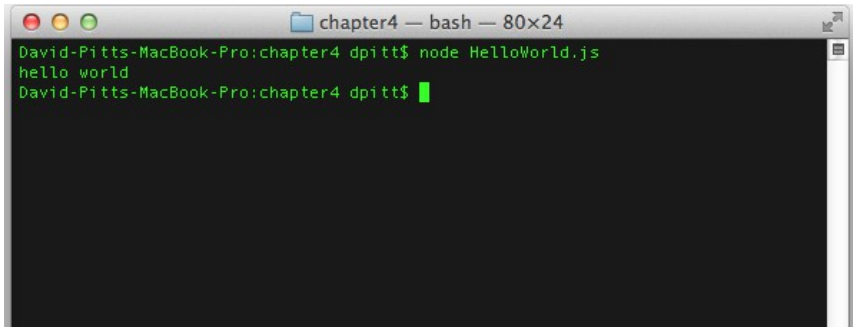

Figure 1 – Executing JavaScript from the command line

2. MODULARITY/STRUCTURE

JavaScript does not have a lot of structural elements like other languages. Part of this is due to its original origins as a dynamic prototype-based language. Modularity is accomplished by partitioning JavaScript functionality into separate files. Typically JavaScript libraries are defined in one giant file which can be painful to maintain and comprehend.

Upcoming tutorials will present modularity workarounds that are necessary for developing large applications with JavaScript, but it is still important to understand primitive JavaScript to establish a foundation of understanding.

Memory

Like other object oriented languages, developers don't specifically need to worry about or perform allocation and deallocation of memory. Since everything is an object that is dynamically created, the runtime environment will utilize a garbage collection mechanism to reclaim objects that are no longer visible or reachable by the current execution context.

In theory, developers should not to worry about memory reclaiming or leaks. However, there are ways that object references become zombied or unreachable by the garbage collector. Closures are one way object references can become unreachable causing a memory leak.

Global Variables

When a page with JavaScript is loaded, objects and variables created and defined with the page consume memory. The garbage collector will track and reclaim memory by objects that are no longer referenced by anything. However there is a way for global objects to be defined that is visible during the lifetime of the browser executable that JavaScript is executing within. A runtime window variable is visible that references a globally available object. You can freely add/attach objects to the window variable. The following code shows an example of a global variable definition.

```
window.userId = "jdoe";                    ← Global Variable

var userId = "jdoe;                        ← Local Variable
```

It's important to note that if you define a variable without VAR, then it's attached to the global window object. The following code shows this:

```
userId = "jdoe";                           ← Attached to Window
```

BEST PRACTICE: You should rarely need to define global variables by attaching to the window property. For a Single Page Application, most frameworks will provide a pattern for defining global objects.

Whitespace and Semicolons

Previous tutorials have described the origins of JavaScript and pointed out its features as a dynamic object-based language. The name "JavaScript" likely came to be due the similarity with the Java syntax. Like Java, JavaScript syntax is simple, free form, and case sensitive. Expressions are terminated with a semicolons.

Listing 2 – An example of JavaScript syntax

```
var abc    =    'a'  +    'b' + 'c';
var def =
            'd'   +
         'e'   +
         'f';

console.log(abc + def);
```

Semicolons are required to terminate expressions, but JavaScript cuts slack to lazy developers who forget to terminate their expressions with semicolons. However, it's best practice to always terminate expressions with a semicolon.

Comments

Comments are non executable lines of code that can be applied to help document your code. Block and line comments can be defined.

```
/*
  Block comments
*/

...

// line comments

...

var a = "abc"; // end of line comment
```

Arithmetic Operators

Available operators are as you would expect for arithmetic operations (+, -, /, %). The + is overloaded to support string concatenation. Unary increment and decrement operators are supported in the same fashion as C, C#, and Java. Listing 3 shows some example operators in action.

Listing 3 – An example of arithmetic operators

```
var count = 5;                                    ← Increment/decrement
console.log( --count ); // logs 4
console.log( ++count ); // logs 5
console.log( count-- ); // logs 4
console.log( count++ ); // logs 5

var x = 5;                                        ← Assignment
var y+= 5; // y = 10;
var y-= 5; // y = 5;
var y*= 5; // y = 25;
var y/= 5; // y = 5;
var s1 = "hello";                                 ← String Concatenation
var s2 = "world";
var s3 = s1 + s2;
```

== and ===

The assignment operation == is used for equality checks. It will perform type conversions before checking equality. JavaScript also introduces the === operator which checks for equality. It will not perform type conversion. Study the expressions in listing 4 below and you'll see this nuance.

Listing 4 – An example of ==

```
console.log( 0 == '' );                    ← logs true, performs type conversion
console.log( 0 === '' );                   ← logs false, no type conversion

console.log ( 5 == '5');                   ← logs true, performs type conversion
console.log ( 5 === '5');                  ← logs false, no type conversion

console.log ( true == '1');                ← logs true, performs type conversion
console.log ( true === '1');               ← logs false, no type conversion.
```

BEST PRACTICE: For safeness, always use the === for equality checks.

Flow Control

Execution paths are controlled using if/else and looping commands. They are fairly straight forward, basically the same as you have used in almost all languages.

Listing 5 – An example of execution path control

```
var list = [1,2,3,4,5];
for (item in list) {                       ← For Loop
  console.log(item);
}

for each (item in list) {                  ← Same as for, deprecated, don't use
   console.log(item);
}

for (var i=0;i<list.length;i++)
{
    console.log(list[i]);                  ← For loop with index
}

if (1 + 1 == 2 )  {                        ← if/else
       console.log("The world makes sense...");
 } else {
    console.log("The chaos ensues...");
 }

var count = 10;
while (count > 0) {                        ← Do Loop
   console.log("Count = "+count--);
}
```

Code Blocks

JavaScript expressions can be enclosed code blocks that can be attached to function definitions or defined to delineate conditional and looping expressions. Code blocks are defined using {} characters.

```
function() {...}

If/Else
if (<condition>) {...}

For Loop
for (< expression>) {…}
```

Scope

Scoping around code blocks with JavaScripts differs from what you would expect with other languages. Variables defined within conditional and looping constructs are not locally scoped to an enclosing code block. This is important to understand in order to prevent possible side effects.

What would you expect the console output to be if the expressions that follow are executed?

```
var a = 'abc';

if (true) {
        var a = 'def';
}

console.log(a);
```

You may be surprised to know the output would be `def`, and not `abc`. The variable defined in the conditional code block overrides the outer variable definitions. Scoping behavior within function code blocks behaves as you would expect.

Here is another example. What would you expect the console output to be when these expressions are executed?

```
var a = 'abc';

function s() {
    var a = 'def';
}

s(); A
console.log(a);                    ← Execute function
```

Variable `a` is visible and scoped to function `s()`. So the log output would be `abc;`. Did you answer it correctly?

AMD/CommonJS Module Specifications

Efforts have been made to make JavaScript a viable server side language. Modularity was one area that needed to be addressed, so open source projects created a common module API. One popular open source project is commonJs.org which defines an API for defining module and dependencies. Another popular module API is AMD, which stands for Asynchronous Module Definition. Both have their advantages.

Require.js implements the AMD specification. Loading JavaScript modules asynchronously, instead of synchronously, allows modules to be loaded in an on-demand manner, and helps with performance, debugging an other issues, especially in the browser environment.

The module pattern shown in this section is essentially part of the implementation for used for the specifications. So, in practice you should use an existing proven module dependency framework to support your SPA, but having some understanding of what is going on under the covers is helpful.

3. DATA TYPES

JavaScript has a dynamic type system. This is in contrast to static type languages such a Java and C#, which require data variables to be typed. While this provides type safeness and arguably software that is easier to maintain, it also requires a compilation step, where JavaScript does not, as expressions are interpreted at runtime. There is much debate in the IT community regarding the agility and flexibility of dynamic versus typed languages. Both offer advantages, so you will have to make your own decisions on this debate.

Primitive

Like other object languages, JavaScript provides built-in system primitive types. Primitive types exist for performance, but some primitive types are also defined as objects which allow method calls and can be extended. Like classical languages such as C# and Java, primitive types are passed by value instead of by reference. Since primitive types are typical to most languages, the expression in listing 6 should provide enough usage information.

Listing 6 – Example Primitive Data Types

```
var i = 100;f = 100.10;                    ← Numbers, stored as 64 bit base 10 floating point
var s = 5.98e24;                           ← Very large or small number scientific notation
var s = "hello World";                     ← String, double or single quotes
var b = true;                              ← Boolean
```

BEST PRACTICE: Use a Math library when arithmetic values need to be exact. JavaScript's float has issues (e.g. 0.1 + 0.2 = 0.30000000004) and inexactness in enterprise applications is problematic.

Arrays

Arrays can be created literally or using a constructor approach. Like other languages they are zero-based. Also, since JavaScript is dynamic, initial size does not have to be declared. They just have to be defined. Listing 7 shows some implementation examples:

Listing 7 – Array implementation example

```
var states = ['ks','mo','ne','co'];
console.log(states);

var countries =
countries[0] = "United States";
countries[1] = "Canada";
console.log(countries);

var mixed = ['text',true,10.00];
console.log(mixed);
```

Array Operations

Besides containing a list of data objects, arrays have methods defined to help manipulate and iterate over their contents. There many methods. Listing 8 shows some interesting ones and how you can iterate over them:

Listing 8 – Array operations

```
var a = ['ks','mo','ne','co'];
var b = ['az','ok','tx'];

console.log( a.concat(b) );                ← New array concatenated

console.log( a.join(b) );                  ← Joins arrays into a string

a.push('me');                              ← Push elements to end of array, returns length

console.log( a.pop() );                    ← Removes element from end of array and returns it

for (var ele in a) {                       ← Loop over array
        console.log(ele);
}
```

Undefined and Null

JavaScript introduces a "not defined" data type. This is not to be confused with a null value, as they are not the same. Again, compiled languages do not need an undefined distinction, since anything not defined will result in a compilation error. Undefined data types are set to variables that do not have a value assigned. The null data type represents a value of null. The snippet below illustrates the differences:

```
var v;
console.log(v);                    ← Undefined
var v = null;
console.log(v);                    ← Null
```

Undefined and null causes confusion as many assume that variables and object properties are automatically assigned a null value when defined. The example above shows that they are assigned an undefined value or type. Since undefined equals nothing, JavaScript provides a shortcut mechanism to check for undefined variables, as with the expression below. This works for undefined and null.

```
var v;
if (v) {console.log{true)}         ← true
var v = null;
if (v) console.log(true);          ← true
```

Since undefined is assigned by default, it's safer to use the shortcut method to check for empty data values, instead of doing null checks, as shown below:

```
var v = null;
if (v == null) { console.log("value is null"); }
```

Why, well what happens if the developer forgets to initialize a data value with null? A bug in logic could occur.

BEST PRACTICE: Don't initialize your variables and properties with null. Rely upon JavaScript undefined default and perform checks using the `if(value)` default value.

4. OBJECTS

Everything in JavaScript is an object. Strings, Numbers, Arrays, and even functions, are objects that have properties and methods. System objects supplied by the runtime environment implements objects for primitive types. They are sometimes called "wrapper objects," as they wrapper their respective primitive data types.

Here are some expressions that send methods calls to some high level primitive objects supplied:

```
var s = 'hello world';                          ← String
console.log(s.length);                          ← Number, Displays 11

var amount = 100.12345;                         ← Number
console.log(amount.toFixed(2));                 ← Number, Displays 100.12
```

Like object oriented languages, JavaScript also has a new operator. It can be used to create primitive object instances.

Listing 9 (located on page 10) shows some examples using the new operator. There's more about the new operator further along in this tutorial, so stay tuned.

Listing 9 – New operators in JavaScript

```
var s = 'hello world'                          ← String
console.log(s.length);                         ← Displays 11

var amount = 100.12345;                        ← Number
console.log(amount.toFixed(2));                ← Displays 100.12

var s = new String('hello world');             ← String Object
console.log(s.length);                         ← Displays 11

var amount = new Number(100.12345);            ← Number Object
console.log(amount.toFixed(2));                ← Displays 100.12

var d = new Date();                            ← Date Object
console.log(d.getMonth());                     ← Displays current date month (0-11)
```

Built-In Objects

Everything in JavaScript is an object provided by the JavaScript runtime environment. Here is a list of available object types:

- String – Array of character values
- Boolean – Conditional true/false
- Date - Represents date time value
- Number – Represents all integral and floating point numeric values
- Math – Provides methods for mathematical functions such as abs, log, tan, etc.
- Function – Executable block of code that can accept parameters and return a value
- Object – Base object prototype for all objects
- RegExp – Perform pattern matching, search, and replace on strings

You've already seen how some of these objects are used for primitive data types and arrays. What may not be obvious is that functions are also objects, or "first-class objects," meaning function objects can be created using JavaScript syntax. More details about function objects are coming up, but first let's jump into some details about JavaScript objects.

Creating Objects

You've seen system objects provided by the JavaScript runtime. Like other object oriented languages, customer or user defined objects can be created and used. However, JavaScript objects differ from classic "class"-based object oriented languages which have inheritance, encapsulation, and polymorphism constructs built into the language. JavaScript objects are dynamic and to support this dynamic behavior, a prototype-based approach is taken to object creation. The next sections should give you a good idea for how this works.

There are two ways to create objects with JavaScript: literally or with a constructor function.

Literal Objects

Literal JavaScript objects are defined using JavaScript Object Notation (JSON). Some may think that JSON is just a data format used for transmitting data from a server or remote system. It is, but its real purpose is to define JavaScript objects that have properties and executable methods, or actually functions.

As an example, consider an object that models an account with an ID, name, and balance properties, with methods that debit and credit the account. Figure 2 (located on page 11) shows an account object model and source code using JSON for its Javascript implementation.

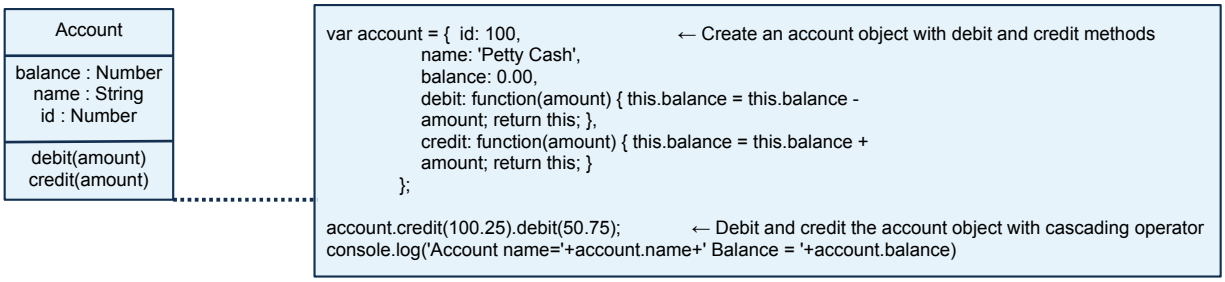

Figure 2 – Petty Cash account object model and source code using JSON

Additionally, Listing 10 below shows a "literal" account object definition with debit and credit methods and relevant properties:

Listing 10 – Literal Account Object Definition

```
var account = {  id: 100,
                 name: 'Petty Cash',
                 balance: 0.00,
                 debit: function(amount) { this.balance = this.balance - amount; return this; },
                 credit: function(amount) { this.balance = this.balance + amount; return this; }
              };
```

Notice in the listing above that the account object has properties representing account ID, balance, and name, along with methods defined for debiting and crediting. Methods are evaluated against the object instance using the "." operator. Notice how methods can be cascaded since the method implementations return `this`**.** The expression below shows the cascading debit and credit calls to the account object:

```
account.credit(100.25).debit(50.75);
console.log('Account name='+account.name+' Balance = '+account.balance);
```

The dynamic nature of JavaScript can be seen when adding new properties or methods, as you simply add them to the object. Here's how a `close()` account method can be dynamically added to the account object:

```
account.close = function() { this.balance = 0;};          ← Add new close function to account object
account.credit(100.25).debit(50.75);                                ←  Debit/credit account
console.log('Account name='+account.name+' Balance = '+account.balance);
account.close();                                                          ← Close the account
```

All Objects are instances of the JavaScript system Object type, and is simply comprised of an associative array and a prototype. We'll talk more about that later. Listing 11 provides some insight into how properties of an object are stored an associative array:

Listing 11 – Properties stored as an associated array

```
var object = { x: 100,                          ← Literal object
               y: 200,
               add: function() { console.log(this.x + this.y)}
               }

for (key  in object) {
        console.log(key);                      ← Outputs property names x y add to the console
}
```

Property elements can be accessed using the property name. Listing 12 (located on page 12) shows a literal object created as a mechanism to map state abbreviations to state name.

Listing 12 – An example of using array subscript syntax

```
var map = { ks: "Kansas",                          ← Object Map of States
            mo: "Missouri",
            ca: "California"
          };

 console.log(map["ks"]);                           ← Access by key, outputs Kansas
```

Notice how the object property is accessed using array type access, but instead of an index number the name of the property is specified.

Constructor Function Objects

An alternative way to define and create an object is referred to as an object constructor. This approach feels a little more like the classic approach, as the classical "new" operator is used to create an instance. Also, constructor objects allow an instance to be initialized with supplied values.

In listing 13, the account object is defined and used with the constructor approach. Notice how the initial balance is initialized and the instance is created with the new operator.

Listing 13 – An example of object constructor

```
var Account =                                      ← Constructor account object
           function(initialBalance) {
                 this.id = 200,
                 this.name = 'R&D',
                 this.balance = initialBalance
                 this.debit =       function(amount) { this.balance =
              this.balance - amount; return this; },
                 this.credit = function(amount) { this.balance = this.balance
              + amount; return this; }
            };

var object = new Account(5123.25);                 ← Create instance
console.log('Account name = '+object.name+' Balance = '+object.balance);
```

BEST PRACTICE: Variables referencing constructor functions are typically camel-cased and with the first character capitalized. Create a factory that hides the "new" keyword since it's easy to forget.

Functions are discussed in an upcoming section. But, as you have seen constructor objects look like functions, look closely. They really aren't functions; they provide a way to initialize objects, enclose the structure of an object, and provide a way to "construct" objects when needed. This is as opposed to defining them literally.

Prototypes

Now that we have explored a couple of ways to define and create objects, let's dive under the covers and see whats going on with objects.

JavaScript is referred to as a Prototype-based language. This can be contrasted to the classic class-based languages in which classes contain methods and properties are defined. At runtime, class meta data is turned into a type system object model. However, available classes must be defined at construction time. JavaScript's dynamic nature applies a prototyped-based approach to objects. As indicated, JavaScript objects are instances of an object with an associative array (key/value) of other data objects or function objects. And, as we have seen, objects can be created at runtime, with methods and properties being added at will.

Every constructor-based object definition has a prototype property that points to the same prototype of the constructor function. Adding a new method to the constructor object is visible to all instances that have been created from it, as shown in listing 14 located on page 13.

Listing 14 – Adding a new method to the prototype

```
var Name =  function() {                          ← Name constructor object definition
                this.first = null,
                this.middle = null,
                this.last = null};

  var nameA = new Name();                         ← Instance is created
  nameA.first = 'Jane';
  nameA.last = 'Doe';

  var nameB = new Name();                         ← Another instance is created
  nameB.first = 'John';
  nameB.last = 'Doe';

  Name.prototype.middle = 'Chris':                ← Middle name added to Name prototype

  Console.log(nameA.middle);                      ← Middle name is visible to both instances
  Console.log(nameB.middle);
```

Listing 14 showed a definition of a constructor function object named `Name`. It has two properties: `first` and `last` name. Two instances are created using the `new` operator and properties are set. Notice these two instances reference the constructor objects prototype property. Then a middle name property is added to the prototype reference. This makes the middle-name property visible to both instances, `nameA` and `nameB`.

Prototype Chaining/Inheritance

The previous section showed how methods and properties that need to be shared across all instances can be made global to all instances by adding properties to the constructor objects prototype. When a property or method is sent to an object during execution, the runtime environment will look for the property/method in the current instance, then in the existing prototype and up the chain until `Object.prototype` is encountered. This is shown in figure 3.

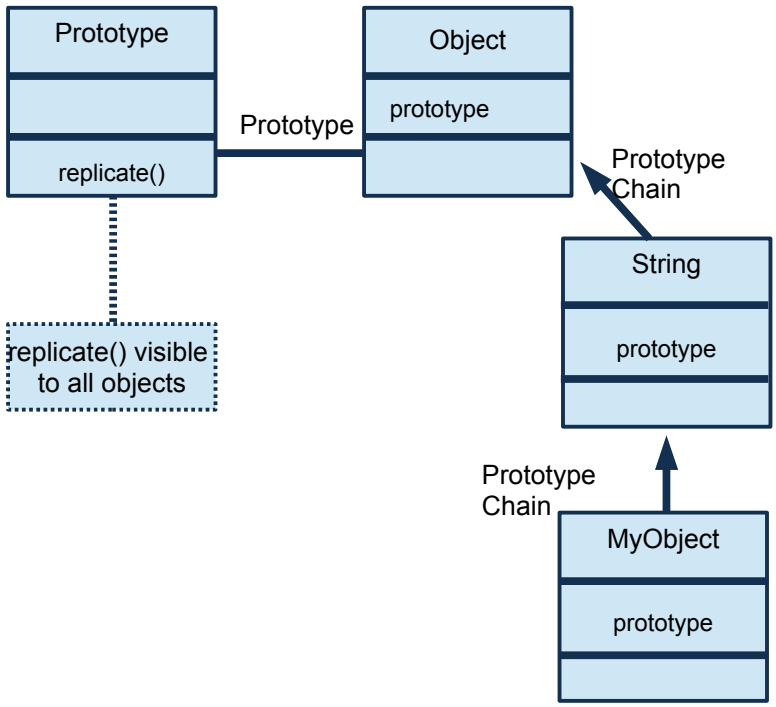

Figure 3 – Prototype Chaining Inheritance

This is referred to as prototype chaining and is how inheritance works in JavaScript. The next expression (located on page 14) demonstrates this chaining behavior by adding a `replicate()` method to the built-in Objects prototype property.

```
Object.prototype.repeat = function(count) {                    ← All objects now have
    return count < 1 ?  '' : new Array(count + 1).join(this);      replicate() method
};

console.log("Hello".repeat(3));                                ← logs "HelloHelloHello";
```

This new `replicate()` is now visible to all objects. While it works for String objects, it will probably throw an error trying to execute when run against other objects since the `split()` method is expecting a String object. So, adding the `replicate()` method to the String objects prototype makes it visible to only String objects.

```
String.prototype.repeat = function(count) {                    ← All String objects now have
    return count < 1 ?  '' : new Array(count + 1).join(this);      replicate() method
};

console.log("Hello".repeat(3));                                ← logs "HelloHelloHello";
```

As you can see, this is similar to inheritance. Let's apply this to an anonymous function, shown with these expressions in listing 15:

Listing 15 – Prototype chaining inheritance with an anonymous function

```
 var XY = function() { this.x = 100,
                       this.y = 200 };

var a = new XY();                                              ← #A Create instance
a.multiply = function() { return this.x * this.y;};            ← #B Add multiply method
console.log(a.multiply());                                     ← logs 2000
var b = new XY();                                              ← #C Create another instance
console.log(b.multiply());                                     ← #D Error multiply() is not defined

XY.prototype.multiply = function() { return this.x * this.y;}; ← #E Add multiply to XY prototype
console.log(b.multiply());                                     ← #F Multiply is available and
                                                                   displays 20000
```

The example defined an XY constructor object **#A**, then assigned a multiply function **#B** and executes it. Then another XY instance is created **#C** and the multiply method is issued, but it is undefined **#D**. This is because the `multiply()` method was added to an object, but not its prototype. Adding the multiply function the XY constructor functions objects prototype **#E**, makes it available to all instances created from the XY object **#F**.

PROTOTYPES IN ACTION – IMPLEMENTING THE SINGLETON PATTERN

Prototype behavior can be seen in action when trying to implement the singleton pattern with JavaScript. Singletons are a pattern seen commonly in classical object languages. Its intent is to implement a global single instance of an object. With JavaScript it is easy, too easy, to make an object a global variable in JavaScript. Simply set an object reference to the globally visible window object. Here is how the current user of an application can be made global:

```
CurrentUser = {userId: 'jdoe', name:'John Doe' };
```

or

```
window.CurrentUser = {userId: 'jdoe', name:'John Doe' };
```

For a large application, putting this global module definition in its own JavaScript file will help make things more modular and maintainable. However, if this file is loaded or referenced multiple times, it will wipe out previous values. So the single pattern can be applied to ensure only a single instance of an object exists no matter how many times this file is loaded. The singleton pattern for a global `CurrentUser` is shown in listing 16.

Listing 16 – Singleton instance implementation

```
var CurrentUser =  function() {
        var User = function() {
                var userid = '';
                var name = '';

                return {

                        getName : function() {
                                return name;
                        },
                        getUserId : function() {
                                return userid;
                        },
                        setName : function(newName) {
                                name = newName;
                        },
                        setUserId : function(newUserId) {
                                userid = newUserId;
                        },
                };
        };
        if (User.prototype._instance) {
                return User.prototype._instance;
        }
        User.prototype._instance = new User;
        return User.prototype._instance;

}();
                                                        ← set singleton value
CurrentUser.setUserId("jdoe");
CurrentUser.setName("John Doe");

console.log(CurrentUser.getUserId());
```

JavaScript prototype behavior provides a convenient way to enforce the singleton instance of a current user no matter how many times the file is loaded. A closure is defined that returns the singleton instance. The singleton instance is set to the constructor functions prototype for the first request and subsequent calls just return the original instance.

Also, notice how getter/setter access methods were defined, as this is a practice not typically done in JavaScript OO development. But, for this example, it shows how access to the data can be encapsulated with methods.

5. FUNCTIONS

Everything in JavaScript is an object, and functions are no exception. JavaScript functions represent a modular unit of execution and are considered first class objects, as they can be created literally, dynamically, assigned to variables, and passed around as data. Literal functions are something you have already seen in this tutorial. Here's a basic literal function definition you've probably seen before:

```
function helloWorld() {
    console.log("hello world");
}
                                                    ← execute function
helloWorld();
```

Literal functions are akin to method implementations in classic languages like Java and C#. JavaScript being dynamic in nature does not really have much in common with compiled-based languages. Comparisons are probably made to Java due to the "Java" in "JavaScript."

One advantage that dynamic-based languages like JavaScript have over their compiler-based competitors is that functional programming capabilities are made possible by the ability to treat chunks of code like data. This can lead to elegant designs that do more with less code.

Anonymous/Closures

Anonymous functions or closures are a powerful element in JavaScript. Closures are a key element to functional programming techniques. Other languages such as C# provide closures. Java has been promising closures for a number of releases, but has yet to provide this capability.

Closure functions are defined and assigned to a variable that can be passed around just like a piece of data, and then executed. You'll see closures commonly used to provide callback and event handling functionality. Functions can be defined and assigned to a variable that can then be passed around and executed.

Check out listing 17 below, as this example contrasts a literal function with one that is created anonymously, or as a closure.

Listing 17 – Literal function contrasted with closure

```
function helloWorld() {                              ← Literal function definition
    console.log("hello world");
}

helloWorld();                                        ← Execute function, outputs "hello world"
var hello = function() { console.log("hello"); };    ← Define anonymous function and assign to variable
hello();                                             ← Execute function, outputs "hello"

var log = function(text) { console.log(text); };     ← Define anonymous function with argument
log("Hello World");                                  ← Execute function, outputs "Hello World"
```

Let's make things a little more interesting with the next example. These expressions define an anonymous function that is passed into another function and then executed. Notice how arguments are handled in the following expressions:

```
var hello = function() { console.log("hello"); };        ← define hello function
var executor = function(func) { console.log(func()); };  ← define executor function

executor(hello);                                         ← invoke executor function, pass in hello
                                                           function as an argument, outputs 'hello'
```

Memoizing

Since functions can be treated as data, an interesting feature becomes available, referred to as "memoization." This feature provides the ability to hide or remember data. Variables scoped by an outer function and referenced by an inner function remember their values every time the function is invoked.

Memoization can be seen in listing 18 (located on page 17). This example implements a literal function that returns anonymous functions for a specific operation. Each operation function can then be executed and results returned. Notice how the sum variable is remembered between operation calls.

Listing 18 – Memoization Closure Example

```
function operationFactory(operation,initialValue) {      ← #1 function that returns operation
                                                            closure function and sets initial value
  var sum = initialValue;                                ← sum variable will be "memoized" by
                                                            operation closure functions
  if (operation == "+")
     { return function(x) { sum += x;return sum;} };
  if (operation == "-")
     { return function(x) { sum -= x;return sum;} };
  if (operation == "*")
     { return function(x) { sum *= x;return sum;} };
}

var add = operationFactory("+",0);                       ← #2 get operation functions from factory
var subtract = operationFactory("-",200);                     and assign to variable
var multiply = operationFactory("*",10.0);

add(100);                                          ← #3 execute add function, sum variable will be 100
console.log(add(200));                             ← #4 add 200, output will be 300, as the previous
                                                        add variable is remembered
subtract(100);                                     ← console output will be 50
console.log(subtract(50));

multiply(0.5);                                     ← console output will be 2.5
console.log(multiply(0.5));
```

Let's walk through code listing, as this is an important concept to see in action. Step **#1** defines an `operationFactory` function that accepts an operation identifier and an initial value. When called, a closure is returned that performs an arithmetic operation against the outer functions `sum` variable. Step **#2** gets operation closure functions from the factory and assigns them to variables. Step **#3** then invokes the function with a value of 100. Step **#4** invokes the add function again with 200 and outputs 300 to the log.

Since the add function was executed to separate times, you might think that the second add execution would output 200. Closure memoization remembers the sum variable across executions of the same function instance.

What do you think the output of the expression shown below will be?

```
var add = operationFactory("+",100);
add(50);
add(50);
console.log(add(50));
```

If you guessed 250, then you are right. The initial value of the "memoized" sum variable is set to 100.

Execution Context

This is a concept that causes confusion, especially with closures. Classical object oriented developers understand the concept of the `this` keyword, which provides a way to reference an existing object reference. This is especially useful when having to access properties/methods and in passing object references around.

However, in JavaScript "this" may not be the "this" you were expecting. Since JavaScript is dynamic code, it has the concept of an execution stack, and since JavaScript runs on a single thread, only one code block is visible in the execution context. "this" is referencing that execution context.

Here are some common JavaScript execution contexts for the `this` operator:

- Window
- Document
- Function
- Method
- Constructor Method

Here is where context problems often occur. Say you are defining a literal object method that defines a closure that performs a calculation and prints results when a button is clicked. It could look something like the source below:

```
var add =  {                                                    ← Literal object
     sum: 0,
     execute: function(x,y) {this.sum =  x + y; }               ← this is in object method context
     print: function() {
         var btn = document.getElementById("print_button");     ← Get document button object
         btn.onClick = function() {
             this.execute(100,100):  ←  WILL FAIL, why?...context, or this will be in document context
             alert(this.sum);        ←  WILL FAIL, why?...context or this will be in document context
          }
        }
   }
```

However, this will fail when the button is clicked with an error indicating that `this.execute()` and `this.sum` are undefined. Why? Because when the closure function is executed, this will be in the HTML document context.

How can this be fixed? Memoization is the answer. The correct context is preserved and referenced in the closure by defining a variable that references `this`. This variable is then referenced by the closure, preserving correct context reference, and not using the current context of `this`.

Here is the same JavaScript snippet that works. Notice how the this context reference is memoized in the closure:

```
var add =  {                                                    ← Literal object
     sum: 0,
     execute: function(x,y) {this.sum =  x + y; }               ← this is in object method context
     print: function() {

         var btn = document.getElementById("print_button");     ← Get document button object

         var _this = this;                                      ← Reference to this method context

         btn.onClick = function() {
             _this.execute(100,100):                ←  Won't fail, memoized _this is correct context
             alert(_this.sum);                      ←  Won't fail, memoized _this is correct context
          }
        }
   }
```

You can also supply an execution context. It can also be specified and supplied to a function using the function call or apply methods.

Here's an example:

```
function add(a,b) {                                 ← Function definition
         return this.x + this.y + a + b;
}
var o = {x:100, y:100};                             ← Object definition

console.log(add.call(o,200,200));                   ← Invoke function with call, specifying a
                                                      context for this. Outputs 600 to console.
```

This is something that you will encounter often, especially in SPA development, when you will be writing a lot of JavaScript client logic to create user interface elements, and for reacting to events in the HTML document context.

Function Closures in Action and Modularity Support

Modularity and dependency injection are not mechanisms built into JavaScript. With JavaScript, you can use folders/files to help modularize code. Compare this with other languages; enforcing modularity in Java is accomplished using package definitions, while C# uses Namespaces. As dependency injection is not a part of

any language, frameworks have filled the gap. This is changing as both C# and Java have indicated a future in implementing built-in dependency injection mechanisms in their language specifications.

Modularity and dependency injection are key for managing SPA like applications that have rich user interaction requirements. However, the goal of this section is to understand JavaScript functions and closures. Understanding how to apply modularity will help with this goal. Figure 4 illustrates the concept of modularity and dependency injection, showing how modules can be used and dependent modules injected:

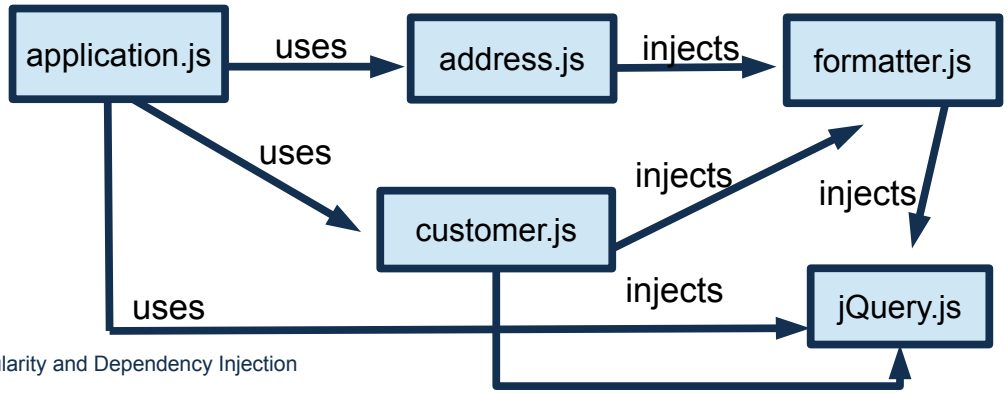

Figure 4 – Modularity and Dependency Injection

Available JavaScript modularity mechanisms are functions or JavaScript code defined in separate files that are loaded using a `<script>` tag. Modularity allows complexity and information to be hidden from consumers of a module. Classical OO languages provide language access visibility to methods and properties of objects. Access modifiers are available and can be specified to make properties and methods private. Private access modifiers prevents developers from changing or accessing elements that are not a part of a module's public access API. JavaScript does have an access modifier, however a pattern has been invented that allows methods and attributes to be hidden.

The module pattern evaluates a function closure that returns a literal object function methods that accesses the private data. Listing 19 shows how an address object is modularized:

Listing 19 – Module Pattern for address

```
var addressModule = (function () {                          ← Address Module reference to object literal
  var address = ['street','city','state','zip',];           ← Array holds address segments (street, city,
  return {                                                    state, zip, etc.)
   street: function (street) {
         address[0] = street;
      return this;
    },
    city: function(city)  {
         address[1] = city;
         return this;
   },
   state: function(state) {
         address[2] = state;
         return this;
   },
   zip: function(zip) {
          address[3] = zip;
          return this;
   },
   format: function () {
      return address[0]+"\n"+address[1]+','+address[2]+'  '+address[3];
    }
  };
})();                                                       ← Function closure evaluated on load
addressModule.street('123 Easy Street').city('Lawrence').state('KS').zip('123456');  ← Invokes module
                                                                                       methods
console.log(addressModule.format());                        ← Outputs formatted address to console
```

Notice how in listing 19, the address elements are stored in an array. Methods are defined to allow array elements to be set, and a method to return a formatted address is defined. Only the methods returned by the literal object returned are visible to the user of the address object. This pattern effectively makes the address array visible or private to the returned literal object.

Dependency Injection

Another pattern commonly found in classical languages is dependency injection. This is simply a pattern for referencing other "dependent" modules. Doing this in a consistent manner communicates dependent modules and provides a way to report or assert modules that are not present, which can help with maintenance and debugging. Listing 20 shows how the previous module pattern introduces a module that formats addresses:

Listing 20 – Injecting a dependent address module

```
var addressModule = (function (printer) {                ← Printer module supplied to address module
  var address = ['street','city','state','zip',];
  return {
   street: function (street) {
         address[0] = street;
      return this;
    },
    city: function(city)  {
         address[1] = city;
         return this;
   },
   state: function(state) {
         address[2] = state;
         return this;
   },
   zip: function(zip) {
          address[3] = zip;
          return this;
   },
   format: function () {
      return printer.format(address);                    ← Engage printer module and format address
    }
  };
})(printerModule);                                       ← printer module reference, assume this is a
                                                           global variable
addressModule.street('123 Easy Street').city('Lawrence').state('KS').zip('123456');
                                                         ← check the counter value and reset, Outputs: 1
console.log(addressModule.format());
```

The address module is "injected" as an argument in module closure, and a reference to the injected module(s) are applied in the module loaded function call.

6. EXCEPTIONS / ERRORS

When errors occur during JavaScript execution, an error object is thrown. You may be surprised to know that problems can be caught by errors being thrown or raised. Exceptions are an integral part of Java, C#, and C++ languages. You don't see a lot of exception handling code in JavaScript. Typed languages mentioned above have the advantage of having exception types that can provide additional debugging information as to why the exception occurred.

Exceptions in JavaScript are actually errors that have occurred during execution. Let's say you want to catch an undefined error, here's how this is accomplished with a **try/catch** code block:

```
try {
       var x = 0;
       var z = x + y;

  } catch (error) {                                      ← Y not being defined will throw an exception
     console.log("Y is not defined, you big dummy :)");  ← catch block outputs message to console
  }
```

You can also throw or raise errors in your code using the ***throw*** clause. You can throw any object type and this instance will be available in the ***catch*** block. Here are some examples of throwing an error with various object types:

```
Throw -1;                                                ← throw -1 number
throw 'Error Message';                                   ← throw error message string
throw {code: 100, message: 'error message' );            ← throw object literal instance with error
                                                           information
```

Catch blocks will have access to the object instances that are thrown.

AJAX

Asynchronous JavaScript and XML (AJAX) is a technology supported by all browsers and is a simple mechanism that provides a profoundly improved user experience. Before AJAX, browsers and JavaScript code would be executed whenever an HTML page was requested from the web server. Then the browser, along with JavaScript, would render an HTML user interface.

AJAX provides a way to request XML or String data from the web server and then process this data with JavaScript. Being able to update individual HTML elements at any granularity likely started the movement towards the SPA applications we see today. You are probably already using AJAX if you are developing web applications using JEE or .NET server side MVC frameworks. Many UI components leverage AJAX to provide a more responsive user interface.

Let's dive into how an AJAX request is made from JavaScript and a response is processed. You've probably noticed the XML emphasis with AJAX. This is due to expecting an AJAX server request to return an HTML document object model (DOM), which is in XML format. Listing 21 shows JavaScript that makes an AJAX call to return an HTML/XML document from the web server:

Listing 21 – XML AJAX request

```
var xmlhttp = new XMLHttpRequest();                      ← Create request instance

xmlhttp.onreadystatechange=function()                    ← Process server side results from call
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    document.getElementById("myDiv").innerHTML=xmlhttp.responseXML; ← XML response
    }
  }

xmlhttp.open("GET","info.html",true);                     ← Server side resource to access
xmlhttp.send();                                           ← Initiate AJAX Request
```

The `xmlhttp` is a global object provided by the JavaScript runtime. The open method specifies put, post, or get operation, the file to open, and returns from the server true if this an asynchronous call, which makes it AJAX. Otherwise it's a synchronous call. A callback function assigned to the `onreadystatechange` property will process the results.

AJAX requests can also return string values from the server instead of XML. Since a JSON string is easily turned into actual JSON with JavaScript, SPA applications will typically utilize server URL requests that return JSON strings containing only application data. SPA applications produce HTML on the client side with JavaScript, so JSON data will be merged with client side dynamic HTML.

Listing 22 (located on page 22) shows how JavaScript invokes a server side URL that returns a JSON string and then turns the string into a JSON object.

Listing 22 – JSON AJAX Request

```
var xmlhttp = new XMLHttpRequest();                    ← Create request instance

xmlhttp.onreadystatechange=function()                  ← Process server side results from call
  {
  if (xmlhttp.readyState==4 && xmlhttp.status==200)
    {
    var json = JSON.parse(xmlhttp.responseText);       ← Parse JSON Text to JSON
    }
  }

xmlhttp.open("GET","info.do",true);                    ← Server side resource to access
xmlhttp.send(); ← Initiate AJAX Request
```

The mechanism of AJAX is supported by all modern browsers and is essential to SPA-based applications. Upcoming tutorials will introduce SPA JavaScript frameworks. One mechanism some of these frameworks implement is a way to access server side data in a RESTful manner. AJAX allows these frameworks to access server side data then update a portion of the user interface, with no page refresh, reinforcing a Rich User Interface.

7. SUMMARY

This tutorial introduced beginning and advanced JavaScript programming features and concepts. A thorough understanding of these topics is necessary for building web SPA applications, as much JavaScript will be developed, used, and applied throughout that process.

If some concepts are still fuzzy, I recommend that you play around with and modify some of the samples to see if it helps your understanding.

References

- Osmani, Addy. *Learning JavaScript Design Patterns.* Volume 1.5.2 http://addyosmani.com/resources/essentialjsdesignpatterns/book/
- Mozilla Developer Network. https://developer.mozilla.org/en-US/
- W3 Schools. http://www.w3schools.com/
- CommonJS Wiki Community. http://wiki.commonjs.org/wiki/CommonJS
- Asynchronous Module Definition (n.d.) In *Wikipedia*. http://en.wikipedia.org/wiki/Asynchronous_module_definition

About The Author

David Pitt is a Sr. Solutions Architect and Managing Partner of Keyhole Software with nearly 25 years IT experience. Since 1999, he has been leading and mentoring development teams with software development utilizing Java (JEE) and .NET (C#) based technologies. Most recently, David has been helping organizations to make the architecture shift to JavaScript/HTML5 and use best practices to create rich client and single page applications.

About Keyhole Software

Keyhole Software is a Midwest-based software development and consulting firm with specialization in Java, JavaScript and .NET technologies. Experts in application development, technical mentoring, and the integration of enterprise-level solutions, Keyhole was founded on the principle of delivering quality solutions through a talented technical team.

Keyhole Software JavaScript Services

- **Outsourced Development** – A Keyhole team provided to perform analysis, design, development, testing and deployment of JavaScript-based SPA applications
- **Development Support** – Specialized members of our team participate as project team member and perform development activities
- **HTML5 / Javascript Education** – 2-day custom course to teach your team the ins and outs of effective enterprise development with JavaScript
- **Mentoring / Player Coaching** – Coaching and knowledge transfer, working with your team to help them understand, use and know best practices in JavaScript and SPA development

For More Information

Keyhole Corporate Kansas City
8900 State Line Road, Suite 455
Leawood, KS 66206
Tel: (877) 521-7769

Keyhole St. Louis
Phone: (314) 329-1699

Keyhole Chicago
200 E Randolph St
Chicago, IL 60601
Phone: (630) 460-8317

TUTORIAL PUBLISHED: OCTOBER 18, 2013