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
```

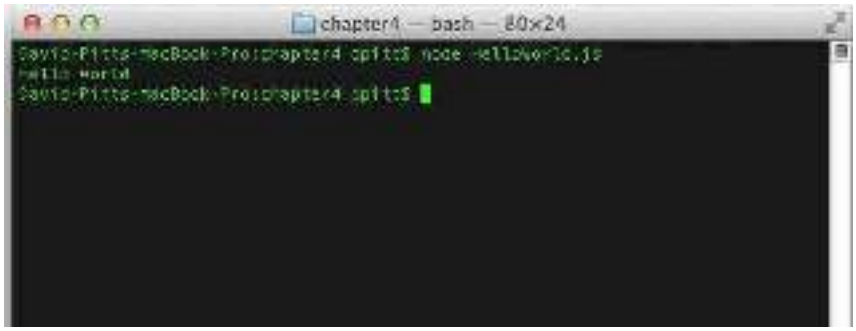

Figure 1 – Executing JavaScript from the command line

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>) {…}
```