JavaScript

Friday, February 20, 2015 7:42 AM

Resources

- Learning JavaScript Design Patterns
- JavaScript, the good parts
- Secrets of the JavaScript Ninja
- Best resources to learn JavaScript
- 60 minute overview of what's coming in ECMAscript 6 (JavaSript)

Best Features

- **Functions as first class objects**
  - Lambdas w/ lexical scoping (closures)
    - (sometimes known as static scoping) is a convention used with many programming languages that sets the scope(range of functionality) of a variable so that it may only be called (referenced) from within the block of code in which it is defined.
- **Dynamic objects w/ prototypical inheritance**
  - Objects are class free. Can add a new member to any object by ordinary assignment.
  - Objects can inherent members from another object
  - Save so much memory over using standard classes -- only one copy of the prototype
- **Object literals and array literals**
  - Convenient notation
  - JS literals were the inspiration for JSON

Awful Parts

- **Global Variables**
- **Scope**
  - Developers expect, and get used to, block scope
- **Semicolon Insertion**
  - Read about it here

```
// Returns undefined. No warning
return {
      status: true
    };
//  can be avoided by moving the bracket to the top line
  return {
      status: true
};
```

- **Reserved Words**
  - Most of the words are not used in the language
- **Unicode**
- **Array == 'Object' && Null == 'Object'**
- **Eval()**
  - Don't use this - access to JS compiler and interpreter
  - evaluates a string as though it were an expression and returns a result
  - Improper use of eval opens up your code for injection attacks

    - Debugging can be more challenging (no line numbers, etc.)
- **The Constructor**
    - Fail to use "new" prefix and you clobber the global scope
    - No way of checking for this issue

Objects

- Simple Types
    - Numbers, strings, booleans, null, & undefined

- Object-like (they have methods), but are immutable
    - Numbers, strings, booleans

- **Collections of name / value pairs, having a hidden link to Object.protoype**
    - **Names**: strings, **Values:** strings, numbers, booleans, & objects
- Mutable keyed collections
- Arrays, functions, regular expressions
- Passed about by reference, never copied
- Usually implemented as hashtables, so values can be retrieved quickly
- Properties:
    - Named values on objects
        - **FirstName:** john - FirstName is a property

Prototype

- When making a new object, you can select the obj that should be its prototype
- Attempts to retrieve a property val from object first, and if not present, checks the prototype

Reflection

- Inspect an obj to determine properties it has
- *Typeof*
    - Typeof flight.number //'number'
- *hasOwnProperty*
    - Only checks obj, ignores prototype

Enumeration

- *For in* statement
- Can loop over all of the property names in an obj
- Best to avoid *for in* entirely
    - Instead, make an array containing the names of the properties in correct order

```
var I;
var properties = [
        'first-name',
        'middle-name',
        'last-name',
        'profession'
        ];
for (I = 0, properties.length; I +=1) {
        document.writeln(propoerties[i] + ': ' +
                another_stooge[properties[i]]);
        }
```

- Get the properties we want, without digging up the prototype chain

Delete

- Remove a property from an obj
- Ignore prototype

```
Delete another_stooge.nickname
```

Functions

- Encloses a set of statements, and each function creates a new execution context (this)
- Are objects
- By default, return *undefined*
- Linked to *Function.prototype*
    - Which is linked to *Object.Prototype*
- Two additional properties:
    - Function's context (this)
    - Code that implement's the function's behavior (arguments)
- Also created w/ a *prototype* property
- Independent of an object
- If a value is a function, we consider it a *method*
    - Can access instance variables with *this*
    - Is *on* an object

Literal

- The name (is optional), can be used to call itself recursively

```
var add = function (a, b) {
        Return a + b;
};
```

First Class Citizens

- Can be assigned to variables, passed as arguments, and even returned

```
function outer() {
        return function () {
                console.log("returning a function within a function");
        }
}
```

Closures

- Defining a function within a function

```
function outer() {
        var name = "Bob';
        function inner() {
                Alert(name);    // 'Bob'
        }
        return inner;
}
var something = outer();
something();    // 'Bob'
```

Invocation Patterns

- Method
- Function
- Constructor
- Apply

Method Invocation

- When a function is stored as a property of an obj

```
var myObject = {
        value: 0,
        increment: function ( inc ) {
                this.value += typeof inc === 'number' ? Inc : 1;
        }
};
```

- Functions that get their object context from *this* are **Public Methods**
- Binding of *this* happens at time of invocation

Functional Invocation **(expression** or **statement[declaration])**

- When a function is not the property of an object
- *This* is bound to the global object
- Use the *that = this* work around so that the inner function will have access to *this* via the variable

```
        myObject.double = function() {
                var that = this ; // Workaround

                var helper  = function () {
                        That.value = add(that.value, that.value);
                };
                Helper();
        };
        myObject.double();
```

Express vs Statement[declaration]

- Main difference between an **expression** & a **statement** is the *function name*
    - can be omitted in expressions to create *anonymous* functions
- Crockford prefers **expressions**
    - Clearly recognizable as to what it really is (a var w/ a function value)
    - Not hoisted -- avoids confusion
    - Can immediately invoke an expression
- Function Smackdown: statement vs expression
- **Expressions** were added later to improve the language -- it worked

Statement

```
// What we see
function funcName() {}

// Under the hood
var funcName = function funcName() {}
```

Expression

```
// stores ref to anon function
var funcRef = function () {};

// stores ref to named fun
var fundRef = func funcName() {};
```

Constructor Invocation

- If a function is invoked with the *new* prefix, then a new obj will be created w/ a link to the value of the function's *prototype* member & *this* will be bound to that new object
- *New* also changes the behavior of the *return* statement
- ALWAYS use the *new* prefix
  - Otherwise, constructor may clobber the *this* it was accidentally passed to (usually global obj, window)
- Three use cases:
  - 1: All objects created w/ the same basic structure
  - 2: object is explicitly marked as an instance of (whatever the object is)
  - 3: easily assign specialized methods to the constructor's prototype

```
var red = new Color(255, 0, 0);
var blue = {r:255, g:0, b:0}

// Outputs true
console.log(red instanceof Color);
// Outputs false
console.log(blue instanceof Color);


var Quo = (string) {
        this.status = string;
};
```

- **Not recommended**. A better style lies ahead.

Apply Invocation

- Lets us construct an array of arguments to use to invoke a functions
- Also lets use choose the value of *this*
- Takes two params:
  - First the value that should be bound to *this*
  - Second: array of params

```
var array = [3, 4];
var sum = add.apply(null, array);  // sum is 7

var status {
        Status: 'A-OK'
};

var status = Quo.prototype.get_status.apply(statusObject);
// status is "A-OK'
```

Arguments

- *Arguments* array is ab onus parameter is available to functions when they are invoked

- Possible to write functions that take an unspecified number of params:

```
var sum = function () {
        var I, sum = 0;
        For (I = 0; I < arguments.length; I +=1) {
                Sum += arguments[i];
        }
        Return sum;
}
```

  Not really an array -- it's an array-like object
  Arguments has *length* property, but lacks all of the array methods

Hoisting

- Using a function before you declare it

```
Hoisted();   // logs "foo"

function hoisted() {
        Console.log("foo");
}
```

- Function *expressions* are **not** hoisted though

```
notHoisted();    // Will not work

var notHoisted = function() {
        Console.log("bar");
}
```

Immediately-Invoked Function Expression (IIFE)

- Knows to parse functions as expressions & not as a statement (declaration)
- What's wrong with "Self Executing Anonymous Function?

```
(function(){ /* code */ }()); // Crockford recommends this one
(function(){ /* code */ })(); // But this one works just as well
```

Private Members in JavaScript

- Douglas Crockford's take
- Public vs Private vs Privileged
  - Privileged:
    - Able to access the private variables & methods
    - Accessible to public methods & the outside
    - Possible to delete or replace, but cannot alter it
    - Assigned with *this* within the constructor
    - Can only be made when an object is constructed (same w/ private)

```
function Container(param) {
   function dec() {
        if (secret > 0) {
           secret -= 1;
```

```
                return true;
            } else {
                return false;
            }
        }
    this.member = param;
    var secret     = 3;
    var that       = this;

    // privileged method
    this.service = function () {
            return dec() ? that.member : null;
        };
}
```

- *Service* is available to other objects & methods, but does not allow access to private members
- This is because of *closures*
- Public:
    - Can be added at any time
    - Function that uses ***this*** to access its object

Patterns

Public:

```
function Constructor (…) {
        this.membername = value;
}
Constructor.prototype.membername = value;
```

Private:

```
function Constructor(…){
        Var                    = this
        Var membername = value;

        Function membername(…) {…}
}
```

Privileged

```
function Constructor(…) {
        This.membername = function(…) {…};
}
```

Singletons

- No need to produce a class-like constructor
- Use an object literal
- Can have public and private methods

```
(function(global) {
  "use strict";
  var MySingletonClass = function() {
if ( MySingletonClass.prototype._singletonInstance ) {
     return MySingletonClass.prototype._singletonInstance;
   }
   MySingletonClass.prototype._singletonInstance = this;
this.Foo = function() {
     // ...
   };
  };
var a = new MySingletonClass();
var b = MySingletonClass();
global.result = a === b;
}(window));
console.log(result);
```

This

- Used to make an object available to private methods
- Workaround for an error in ECMAScript which causes *this* to be set incorrectly for inner functions

Closures

Neat way of dealing with the following two realities of JavaScript:

1. **scope** is at the function level, not the block level
2. much of what you do in practice in JavaScript is **asynchronous/event driven.**

- Simply accessing variables outside of your immediate lexical scope creates a closure
  - The local variables for a function — kept alive *after* the function has returned
- Inner function always has access to the vars & parameters of its outer function
  - Have access to the outer function's variable even after the outer function returns
  - Store references to the outer function's variables

"Use Strict"

This strict context prevents certain actions from being taken and throws more exceptions.
Strict mode helps out in a couple ways:

- It catches some common coding bloopers, throwing exceptions.
- It prevents, or throws errors, when relatively "unsafe" actions are taken (such as gaining access to the global object).
- It disables features that are confusing or poorly thought out.

Toolkit

Set of useful functions below:

Object Function

The object function untangles JavaScript's constructor pattern, achieving true prototypal inheritance. It takes an old object as a parameter and returns an empty new object that inherits from the old one. If we attempt to obtain a member from the new object, and it lacks that key, then the old object will supply the member.

```
if (typeof Object.create !== 'function') {
   Object.create = function (o) {
      function F() {}
      F.prototype = o;
      return new F();
   };
}
newObject = Object.create(oldObject);
```

From <http://javascript.crockford.com/prototypal.html>

Method method

```
function.prototype.method = function (name, func) {
   this.prototype[name] = func;
   return this;
};
```

Use this method to add methods in Number, String, Function, Object, Array, RegExp. For example:

```
Number.method('integer', function () {...});
String.method('trim', function () {...});
```

For vs. For In

- Douglas Crockford recommends in JavaScript: The Good Parts (page 24) to avoid using the `for in` statement
- If you use `for in` to loop over property names in an object, the results are not ordered. Worse: You might get unexpected results; it includes members inherited from the prototype chain and the name of methods.
- *for...in* is perfectly appropriate for looping over object properties. It is not appropriate for looping over array elements.

```
for(var name in obj)
{
    if (obj.hasOwnProperty(name))
    {
    }
}
```

jQuery map vs each

- The `each` method is meant to be an immutable iterator, whereas the `map` method can be used as an iterator, but is meant to manipulate the supplied array & return a new array.
- The `each` function returns the **original array** while the *map function* returns a **new array**
- Can also use the map function to remove an item from an array
- Two additional params:
  - Current index & the array

From <http://stackoverflow.com/questions/749084/jquery-map-vs-each>

jQuery callback example

```
var numbers = [1,2,3,4],
 squareNumbers = function (number) {
     return number * number;
 },
 squares = $.map(numbers, squareNumbers);

console.log(squares);//logs [1,4,9,16]
```

Array.prototype.map callback example

```
var numbers = [1,2,3,4],
    squares = numbers.map(squareNumbers);
```

```
console.log(squares);//logs [1,4,9,16]
```

From <http://abdulapopoola.com/2013/12/15/a-peek-into-javascripts-array-prototype-map-and-jquery-map/>

Passing by reference

Say I want to share a web page with you.

Reference

- If I **refer** you to the URL, I'm **passing by reference.** You can use that URL to see the **same web page** I can see. If that page is changed, we both see the changes. If you delete the URL, all you're doing is destroying your reference to that page - you're not deleting the actual page itself.
- The called functions' parameter will be the same as the callers' passed argument (not the value, but the identity - the variable itself
- Variables that hold reference types actually hold a reference to a location in memory (on the heap)

Value

- If I print out the page and give you the printout, I'm **passing by value**. Your page is a disconnected copy of the original. You won't see any subsequent changes, and any changes that you make (e.g. scribbling on your printout) will not show up on the original page. If you destroy the printout, you have actually destroyed **your copy** of the object - but the original web page remains intact.
- The called functions' parameter will be a copy of the callers' passed argument.

From <http://stackoverflow.com/questions/373419/whats-the-difference-between-passing-by-reference-vs-passing-by-value>

Passing by pointer *is* passing by reference - in the example above, the URL is a pointer to the resource
Great tutorial on the two: http://www.leerichardson.com/2007/01/parameter-passing-in-c.html

Setting default parameter values

```
function foo( a, b ) {

 a = a || '123';
 b = b || 55;
 Print( a + ',' + b );
}

foo(); // prints: 123,55
foo('bar'); // prints: bar,55
foo('x', 'y'); // prints x,y
```

Multiple string concatenation

```
var html = ['aaa', 'bbb', 'ccc', ...].join('');
```

The Module Pattern

```
// Create an anonymous function expression that gets invoked immediately,
// and assign its *return value* to a variable. This approach "cuts out the
// middleman" of the named `makeWhatever` function reference.
//
// As explained in the above "important note," even though parens are not
// required around this function expression, they should still be used as a
// matter of convention to help clarify that the variable is being set to
// the function's *result* and not the function itself.

var counter = (function(){
var i = 0;==

return {
        get: function(){
        return i;
        },
        set: function( val ){
        i = val;
        },
        increment: function() {
        return ++i;
        }
};
}());

// `counter` is an object with properties, which in this case happen to be
// methods.

counter.get(); // 0
counter.set( 3 );
counter.increment(); // 4
counter.increment(); // 5

counter.i; // undefined (`i` is not a property of the returned object)
i; // ReferenceError: i is not defined (it only exists inside the closure)
```

Working with unlimited parameters

There's a weird "magic" variable you can reference called "arguments":

It's *like* an array, but it's not an array. In fact it's so weird that you really shouldn't use it much at all. A common practice is to get the values of it into a *real* array:

The simple, straightforward way to turn `arguments` into an array is therefore

```
function foo() {
  var args = [];
```

```
 for (var i = 0; i < arguments.length; ++i) {
         args[i] = arguments[i];
  } // ...
}
```

Source: http://stackoverflow.com/questions/6396046/unlimited-arguments-in-a-javascript-function