Advanced Javascript with jQuery

Chris Langtiw, Training Connection

Advanced Object Concepts

Closures

- Functions have access to the context (scope) in which they are created.
- A closure is created when a scope object persists (is not deallocated) when the context completes execution – usually because of an external reference to the scope object.
- Ideal for protecting and/or hiding data, or making data persistent without relying on global variables.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Demo</title>
</head>
<body>
    <button>Click Me!</button>
    <button>Click Me!</button>
    <button>Click Me!</button>
    <button>Click Me!</button>

<script type="text/javascript">
var butList = document.body.getElementsByTagName('BUTTON');
for (var c = 0; c < butList.length; c++) {
    butList[c].onclick = function() {
     alert(c); // will always return '4' -
               // the value of c AFTER loop completes
    }
}

</script>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Demo</title>
</head>
<body>
    <button>Click Me!</button>
    <button>Click Me!</button>
    <button>Click Me!</button>
    <button>Click Me!</button>

<script type="text/javascript">
var butList = document.body.getElementsByTagName('BUTTON');
function addClickHandler(button, index) {
    button.onclick = function() {
     alert(index);  // Returns the expected index value because
                    // the parent function's scope object persists
    }
}
for (var c = 0; c < butList.length; c++) {
    addClickHandler(butList[c], c);
}
</script>
</body>
</html>
```

```
<script type="text/javascript">
/*
A common pattern is to have a function return a function, allowing us to
encapsulate and protect data. (variation of factory pattern)
Note the use of an anonymous function in an assignment statement for
positioning flexibility.
*/
var butList = document.body.getElementsByTagName('BUTTON');

var makeHandler = function(index) {
    // return the actual handler method
    return function(e) {
     alert(index); // index is persistent and protected
    }
};

for (var c = 0; c < butList.length; c++) {
    butList[c].onclick = makeHandler(c);
}
</script>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Demo</title>
</head>
<body>
    <button>Click Me!</button>
    <button>Click Me!</button>
    <button>Click Me!</button>
    <button>Click Me!</button>
<script src="/lib/jquery-1.9.1.js" type="text/javascript"></script>
<script type="text/javascript">
$('button').each(
    function(index, element) {
     $(element).click(
          function() {
               alert(index);
          }
     );
    }
);
</script>
</body>
</html>
```

```
<script src="/lib/jquery-1.9.1.js" type="text/javascript"></script>
<script type="text/javascript">
/*
Closures allow persistent data that can be changed while protecting
the data from outside tampering
*/
$('button').each(
    function(index, element) {
     var c = 0;     // c remains persistent in each scope object
                    // created at function invocation
                    // c is private and not accessible elsewhere
    $(element).click(
          function() {
               alert('Button ' + index +
                      ' has been clicked ' +
                      (++c) + ' times!');
          }
     );
    }
);

</script>
```

Immediately-Invoked Function Expressions (IIFEs)

- An IIFE is a function declaration written as an expression and then executed at declaration time
- Useful for creating temporary objects or creating protected/hidden namespaces via closures

```
(function() {
   /*
    Items created within this function do not affect outer
    scope and are hidden/protected
   */
})();
```

```
<button id="button1">Click Me</button>

<script type="text/javascript">
// Conventional example using a named function closure
function init() {
    var c = 0;
    document.getElementById('button1').onclick = function() {
     alert('Click count: ' + (++c));
    };
}
init(); // we have to call the function to start the process
</script>

<script type="text/javascript">
// Same thing, only use an IIFE to avoid having to execute the function separately
(function() {
    var c = 0;
    document.getElementById('button1').onclick = function() {
     alert('Click count: ' + (++c));
    };
})(); // our function is run immediately after declaring it
</script>
```

Namespaces

- Used to avoid issues that arise from cluttering the global environment such as name collisions
- Helps to organize and modularize code

```
/*
    Simplest namespace - Single-Object
    Pros - easy to use
    Cons - multiple objects can still clutter global space in large projects,
    objects may still get overwritten without existence check
*/

var Application = {
    /*
     properties and methods go into this object instead of
     the global namespace
    */
};

// safer implementation in larger projects/multiple developers
var Application = Application || {}; // use existing object or create new one
/*
    extending the Application object separately allows us to extend the existing
    object without overwriting it
*/
Application.method = function() { … };
// object can also be further extended into subobject namespaces
Application.subObject = {};
Application.subObject.newProperty = … ;
```

```
// Alternate namespace pattern using IIFE

// src: http://addyosmani.com/blog/essential-js-namespacing/
// namespace (our namespace name) and undefined are passed here
// to ensure 1. namespace can be modified locally and isn't
// overwritten outside of our function context
// 2. the value of undefined is guaranteed as being truly
// undefined. This is to avoid issues with undefined being
// mutable pre-ES5.
(function ( namespace, undefined ) {
    // private properties
    var foo = "foo",
        bar = "bar";
    // public methods and properties
    namespace.foobar = "foobar";
    namespace.sayHello = function () {
        speak("hello world");
    };
    // private method
    function speak(msg) {
        console.log("You said: " + msg);
    };
    // check to evaluate whether 'namespace' exists in the
    // global namespace - if not, assign window.namespace an
    // object literal
})(window.namespace = window.namespace || {});
```

Example: Creating a jQuery Plugin

Key points to remember when creating a plugin:

- Only take up a single namespace in the jQuery object
- Use IIFEs to encapsulate code
- $.method vs. $.fn.method declaration
- Return jQuery collection to maintain chaining
- Remember that you're working with a set of nodes

Object Oriented Programming in Javascript

- Javascript does not have classes – it uses prototypal inheritance
- Objects are based off of other objects (their 'prototype')
- Object constructors are functions
- An instance of an object is created by invoking the constructor using the `new` keyword
- Object literals are derived from the generic `Object` type and have no constructor
- Object properties are public

Object Constructor

- Constructor functions are used to create instances of objects via the `new` keyword
- Properties declared using `this` within the constructor are public and copied to the local instance
- Private variables within the constructor are NOT accessible to the object instances

```
// define a Car template object
var Car = function(make, model) {
    // ‘this’ refers to a specific instance of Car
    this.make = make || null;
    this.model = model || null;
    this.start = function() {
     console.log('started.');
    };
};

// use ‘new’ to create instances of the constructor object
// each instance is a separate object
var myCar = new Car('Chevy','Camaro');

var yourCar = new Car('Ford','Mustang');
```

Object Prototype

- An object ‘inherits’ the properties of objects within its prototype chain
- Local properties will override properties of the same name in the prototype chain
- An object’s prototype object is set at creation time and cannot be changed afterwards
- A constructor can have its prototype altered and the changes will affect existing and future instances
- If the prototype object itself is reset, only future instances will use the new prototype object

```
// define a Car template object
var Car = function(make, model) {
    // ‘this’ refers to a specific instance of Car
    this.make = make || null;
    this.model = model || null;
    this.start = function() {
     console.log('started.');
    };
};

// each instance is a separate object
var myCar = new Car('Chevy','Camaro');

// ALL instances, existing and future, will inherit the stop method
Car.prototype.stop = function() {
    console.log('stopped.');
};

var yourCar = new Car('Ford','Mustang');

myCar.stop(); // works
yourCar.stop() // works
```

```
var Car = function(make, model) {
    this.make = make || 'default Make';
    this.model = model || 'default model’;
}

var SUV = function() {
    this.has4WD = true;
}

var yourCar = new SUV();

/*
Set the prototype of SUV to Car. Any instances of SUV
AFTER setting the prototype will also inherit from Car
*/
SUV.prototype = new Car();

var myCar = new SUV();

console.log(myCar.make); // 'default Make’
console.log(yourCar.make); // undefined

// All instances with Car in its prototype chain will inherit this
Car.prototype.stop = function() {
    console.log('stopped.');
};
```

Constructors and Prototype

- Best practice is to create empty constructor and assign default properties using prototype

```
var Car = function() {};

Car.prototype.make = 'default Make';
Car.prototype.model = 'default Model';
Car.prototype.start = function() { console.log('started.'); };

var myCar = new Car();
console.log(myCar.make); // 'default Make'
console.log(myCar.hasOwnProperty('make')); // false
myCar.make = 'Chevy'; // local property overrides prototype property
console.log(myCar.make); // 'Chevy'
console.log(myCar.hasOwnProperty('make')); // false
```

Object Literals and Prototype

- An object literal's prototype is Object.
- New instances cannot be created using new
- Use Object.create() to create new objects instead of new

```
/*
    Object.create is native in ECMA 5+. Most modern browsers support it.
*/
// polyfill solution
if (typeof Object.create !== 'function') {
    Object.create = function(parentObj) {
     var F = function() {};
     F.prototype = parentObj;
     return new F();
    };
}

var Car = {
    make: 'default Make',
    model: 'default Model',
    start: function() { console.log('started.'); }
};

var myCar = Object.create(Car); // make new object of type Car
myCar.make = 'Chevy';
var yourCar = Object.create(Car);
```

Error Handling

- Error handlers make debugging the application easier
- NOT intended as a preventative measure against errors
- Use throw and try…catch to manage errors
- Use the default Error object or create your own custom error object using Error as the prototype

```
function doStuff() {
    throw new Error("something has happened.");
}

function errorHandler(e) {
    console.log('An error has occurred:',e.message);
}

try {
    doStuff();
} catch (e) {
    errorHandler(e);
}
```

```
function customError(data) {
    this.message = data.message;
    this.otherValue = data.otherValue;
}
customError.prototype = new Error();

function doStuff() {
    throw new customError({ message: "oops!", otherValue: 42 });
}

function errorHandler(e) {
    console.log('An error has occurred:',e.message, e.otherValue);
}

try {
    doStuff();
} catch (e) {
    errorHandler(e);
}
```

When to use error handling

Some rules of thumb:

- Once you've fixed a hard-to-debug error, try to add one or two custom errors that can help you more easily the solve the problem, should it occur again.
- If you're writing code and think, "I hope [something] doesn't happen—that would really mess up this code," then throw an error when that something occurs.
- If you're writing code that will be used by people you don't know, think about how they might incorrectly use the function and throw errors in those cases.

When to use error handling

- Errors should be thrown in the deepest parts of the application stack – libraries, includes, etc.
- Application logic is best suited for try…catch
- Never have an empty catch block – always recover from the error somehow

```
/*****
 * debug is a global object that controls error logging
 * Currently implemented as a wrapper for console.log
 * Allows debug level to be universally set so that console messaging
 * may be turned on or off without having to remove debug message code
 * The on() and off() methods were added to allow the debug messages to
 * be turned on or off directly in the client console
 **/
var debug = {
    DEBUG_LEVEL: 1,

    // debug.log function abstracts console.log away and puts it behind a flag
    log: function() {
     if (!this.DEBUG_LEVEL) return;
     var args = Array.prototype.slice.call(arguments);
     console.log.apply(console, args);
    }, // log

    on: function(level) { this.DEBUG_LEVEL = level || 1; },
    off: function() { this.DEBUG_LEVEL = 0; }

}; // debug
```

Best Practices

- ‘Best’ is sometimes subjective
- Balance between performance and maintainability
- Test on per-case basis
- Remember the saying: “Opinions are like…”

Best Practices

- Use a style guide and be consistent

  Google: http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml

  jQuery: http://contribute.jquery.org/style-guide/js/

  Crockford: http://javascript.crockford.com/code.html
- Maintain loose coupling
- Avoid globals

Best Practices

- Don’t modify objects you don’t own
  - Native objects (String, Object, etc)
  - DOM nodes
  - BOM nodes (window, etc)
  - Library objects
- Don’t override methods
- Don’t add new methods
- Don’t remove existing methods

Best Practices

- Pass parameters to functions as objects to avoid confusion dealing with multiple parameters
- Store globals and object properties as local variables when used repeatedly
- Take advantage of event propagation and filtering
- Minimize the amount of work done inside the catch block of a try…catch by calling an error handler from within the catch block

Asynchronous Javascript and XML (Ajax)

- Ajax is a means of creating a better user experience by removing the need to do complete page reloads
- Utilizes browser objects available since IE 5.5 and Mozilla 1
- Perceived performance is better
- Primary means of implementation is the XMLHttpRequest object

Making an Ajax Request using jQuery

- $.Ajax()
- Most important options
  - url
  - type
  - dataType
  - success

Side-loading Page Content

- Put your most important content in the main part of the page
- Secondary content can be loaded via Ajax request after main content loads and renders

Processing Data From The Server

- Main data transports in use today are JSON and XML
- JSON is natively understood by Javascript
- XML may be traversed as an XML Document Object just like the DOM

Additional Useful Request Parameters

- error – Specifies an error handler to use in case things go wrong
- cache – Browser caching workaround

Sending Data To The Server

- Data should be serialized into a query string and placed inside the data parameter

Application Concerns When Using Ajax

- Usability
- Navigation
- Security
  - Cross-site scripting
  - JSONP callback requests

Title

References

- Maintainable Javascript, Nicholas C. Zakas, O'Reilly Media
- High Performance Javascript, Nicholas C. Zakas, O'Reilly Media
- Javascript: The Good Parts, Douglas Crockford, O'Reilly Media
- Essential Javascript Namespacing Patterns http://addyosmani.com/blog/essential-js-namespacing/
- Design Patterns, GoF, Addison-Wesley
- XMLHttpRequest W3C Specification http://www.w3.org/TR/XMLHttpRequest/