Lecture 6: JavaScript IIFE & Closures

CS472 Web Programming

Maharishi University of Management

Department of Computer Science

Assistant Professor Asaad Saad

Maharishi University of Management - Fairfield, Iowa © 2016

Closures

Closure

A first-class function that binds to free variables that are defined in its execution environment.

Free variable

A variable referred to by a function that is not one of its parameters or local variables.

Bound variable

A free variable that is assigned a reference to a variable in the scope chain.

A closure occurs when a(n inner) function is defined and attaches itself to the free variables from the surrounding environment to "close" up those stray references.

Closure Example

```
var x = 1;

function f() {
        var y = 2;
        var summ = function() {
                        var z = 3;
                        print(x + y + z);
                  }; //  inner function closes over free variables x, y as it is declared
        y = 10;
        return summ;
}

var g = f();
g(); // 1+10+3 is 14
```

Common Closure Bug

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() {
            return i; // closure
                }
};
```

```
console.log(funcs[0]()); // 5
console.log(funcs[1]()); // 5
console.log(funcs[2]()); // 5
console.log(funcs[3]()); // 5
console.log(funcs[4]()); // 5
```

```
var helper = function(m) {
     return function() { return m; }
}
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = helper(i);
};
```

```
console.log(funcs[0]()); // 0
console.log(funcs[1]()); // 1
console.log(funcs[2]()); // 2
console.log(funcs[3]()); // 3
console.log(funcs[4]()); // 4
```

Example of closures being helpful with event handling

```
<p>Some paragraph text</p>
<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-18">18</a>

function makeSizer(size) {
        return function() {
                document.body.style.fontSize = size + 'px';
        };
}

document.getElementById('size-12').onclick = makeSizer(12);
document.getElementById('size-14').onclick = makeSizer(14);
document.getElementById('size-18').onclick = makeSizer(18);
```

Practical uses of closures

- A closure associates data with a function—parallel to properties and methods in OOP.
- Consequently, you can use a closure anywhere you might use an object with a single method.
- An event handler is a single function executed in response to an event.
- Closures also very useful in JavaScript for encapsulation and namespace protection
- Loading event handlers `window.onload`
- IIFE and Module pattern

Encapsulation and namespace protection with closures

- Languages such as Java provide private methods, can only be called by other methods in the same class. JavaScript does not provide this, but possible to emulate with closures.
- JavaScript provides powerful way of managing global namespace,
- Public functions that can access private functions and variables using closures: Module Pattern

Module Pattern

```
(function(params) {
        statements;
})(params);
```

=

```
(function(params) {
        statements;
}(params) );
```

- Declares and immediately calls an anonymous function
- Parens are an expression that wraps a function expression that will be immediately invoked

    "immediately invoked function expression (IIFE)"

- Used to create a new scope and closure around it
- Can help to avoid declaring global variables/functions
- Used by JavaScript libraries to keep global namespace clean

Module Pattern example

```
// old: 3 globals
var count = 0;
function incr(n) {

        count += n;
}
function reset() {
        count = 0;
}
incr(4);
incr(2);
console.log(count);
```

```
// new: 0 globals!
(function() {
var count = 0;
function incr(n) {

        count += n;
}
function reset() {
        count = 0;
}
incr(4);
incr(2);
console.log(count);
})();
```

Avoids common problem with namespace/name collisions

Solving the Closure Bug with Module Pattern

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() {
           return i; // closure
              }
};
```

```
console.log(funcs[0]()); // 5
console.log(funcs[1]()); // 5
console.log(funcs[2]()); // 5
console.log(funcs[3]()); // 5
console.log(funcs[4]()); // 5
```

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function(n) {
       return function() { return n; }
       }(i);
};
```

```
console.log(funcs[0]()); // 0
console.log(funcs[1]()); // 1
console.log(funcs[2]()); // 2
console.log(funcs[3]()); // 3
console.log(funcs[4]()); // 4
```

Main Point

Closures are created whenever an inner function is defined and it closes over its free variables. Closures provide encapsulation of methods and data. Encapsulation promotes self-sufficiency, stability, and re-usability.

Science of Consciousness: Transcendental consciousness provides encapsulation of our Self, pure awareness. This experience promotes self-sufficiency, stability, and adaptability.

JavaScript Objects

- Objects in Javascript are like associative arrays
- The keys can be any string
- You do not need quotes if the key is a valid JavaScript identifier
- Values can be anything, including functions
- You can add keys dynamically using associative array or the . syntax

```
var x = {
       'a': 97,
       'b': 98,
       'mult': function(a, b) {
              return a * b;
              }
       };
```

```
console.log(a.x); //97
console.log(a['x']); //97
a.b = 100;
a['b'] = 200;
var result = x.mult(3,3); // 9
```

Loop Over Object Literal

```
var things = {'a': 97, 'b': 98, 'c': 99 };

for (var key in things) {
      console.log(key + ', ' + things[key]);
}

// a, 97
// b, 98
// c, 99
```

this

`this` object is determined by **HOW** the function is called.

```
function a(){ console.log(this); }
var b = {
      log: function(){
            console.log(this);
      }
}
console.log(this); // this generally is window object
a(); // a() is called by global window object
b.log(); // log() is called by b object
```

Be careful!

```
function a() {
        this.newvariable = 'hello';
}

console.log(newvariable); // ReferenceError: newvariable is not defined(…)
a(); // this = window
console.log(newvariable); // hello
```

Self Pattern – The Problem

```
var a = {
       name: '',
       log: function() {
              this.name = 'Hello';
              console.log(this.name); // "Hello "
              var setFrench = function(newname) {
                                      this.name = newname;
                             }
              setFrench('Bonjour');
              console.log(this.name); // "Hello"
       }
}
a.log();
```

Line 9: calling `setFrench` *from* `window`*!*
Line 7: creating variable `window.name = newname;`

Self Pattern – The Solution

```
var a = {
        name: '',
        log: function() {
            var self = this; // self = a Object
            self.name = 'Hello';
            console.log(self.name); // Hello
            var setFrench = function(newname) {
                                self.name = newname;
                            }
            setFrench('Bonjour');
            console.log(self.name); // Bonjour
        }
}
a.log();
```

Setf Pattern: Inside objects, always create a “self” variable and assign “this” to it. Use “self” anywhere else.

JavaScript Object Notation (JSON)

JSON is a syntax for storing and exchanging data and an efficient alternative to XML

```
{"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]}
```

A name/value pair consists of a field name **(in double quotes)**, followed by a colon, followed by a value.

JSON values can be:

- A number (integer or floating point)
- A string (in double quotes)
- A Boolean (true or false)
- An array (in square brackets)
- An object (in curly braces)
- null

`JSON.parse(string)` converts the given string of JSON data into an equivalent JavaScript object and returns it
`JSON.stringify(object)` converts the given object into a string of JSON data (the opposite of JSON.parse)

Emulating private methods with closures (module pattern)

```
var counter = (function()
        var privateCounter = 0; //private data
        function changeBy(val) { //private inner function
                privateCounter += val;
        }
        return { // three public functions are closures
                increment: function() { changeBy(1); },
                decrement: function() { changeBy(-1); },
                value: function() { return privateCounter; }
        }
})();

alert(counter.value()); /* Alerts 0 */
counter.increment();
counter.increment();
alert(counter.value()); /* Alerts 2 */
counter.decrement();
alert(counter.value()); /* Alerts 1 */
```

Emulating private methods with closures (module pattern)

```
var makeCounter = function()
      var privateCounter = 0; //private data
      function changeBy(val) { //private inner function
            privateCounter += val;
      }
      return { // three public functions are closures
            increment: function() { changeBy(1); },
            decrement: function() { changeBy(-1); },
            value: function() { return privateCounter; }
      }
};
var counter1 = makeCounter();
var counter2 = makeCounter();

alert(counter1.value()); /* Alerts 0 */
counter1.increment();
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

Main Point

Objects are another widely used encapsulation mechanism in JavaScript. They are easily created with object literals. They can dynamically add new properties; behave like associative arrays; must use ‘this’ to refer to properties; and have a prototype property that provides class-like functionality. Science of Consciousness: Transcending is easily achieved through the TM Technique. Similar to the benefits of encapsulation provided by JavaScript objects, the state of restful alertness that results from this experience provides an encapsulation of awareness that protects us from negative external stimuli and enhances our ability to interact in productive manners.

.call() .apply() .bind()

There are many helper methods on the Function object in JavaScript

```
var func2 = func.bind(this); // creates a copy
func.call(this, param1, param2 ...);
func.apply(this, [param1, param2 ...]);
```

Function Invocation Example

```
var me = {
        first: 'Asaad',
        last: 'Saad',
        getFullName: function() {
                return this.first + ' ' + this.last;
        }
}

var log = function(height, weight) { // ‘this’ refers to the invoker
        console.log(this.getFullName() + height + ' ' + weight);
}

var logMe = log.bind(me);
logMe('180cm'); // Asaad Saad 180cm undefined

log.call(me, '180cm', '70kg'); // Asaad Saad 180cm 70kg
log.apply(me, ['180cm', '70kg']); // Asaad Saad 180cm 70kg
```

Function Invocation Example

```
var me = {
        first: 'Asaad',
        last: 'Saad',
        getFullName: function() {
                return this.first + ' ' + this.last;
        }
}

var log = function(height, weight) {
        console.log(this.getFullName() + height + ' ' + weight);
}
```

Anonymous

```
(function(height, weight) {
         console.log(this.getFullName() + height + ' ' + weight);
}).apply(me, ['180cm', '70kg']); // Asaad Saad 180cm 70kg
```

Function Borrowing

```
var me = {
        first: 'Asaad',
        last: 'Saad',
        getFullName: function() {
                return this.first + ' ' + this.last;
        }
}

var you = {
        first: 'George',
        last: 'Saad'
}

console.log(me.getFullName.apply(you));
```

Function Currying

```
function multiply(a, b) {
        return a*b;
}

var multipleByTwo = multiply.bind(this, 2);
console.log(multipleByTwo(4));

var multipleByThree = multiply.bind(this, 3);
console.log(multipleByThree(4));
```

Inheritance

JavaScript supports prototype inheritance. So objects get access to properties and methods of their prototype objects.

`Object` is the end of the prototype chain.

```
// a.__proto__ is Object
var a = {};

// b.__proto__ is function
// b.__proto__.__proto__ is Object
var b = function(){};

// c.__proto__ is array
// c.__proto__.__proto__ is Object
var c = [];
```

Function Constructors

- It’s a Function used to create/construct other Objects and doesn’t return a value.
- By convention Function Constructors start with a Capital letter.
- To create new object from a Function Constructor we use the **`new`** keyword. it will invoke the function next to it, copy **`this`** variable and replace it with the new empty created object (because it’s the invoker/creator).
- A property called **`prototype`** in the constructor is used to extend/add new functionalities to all objects created by the constructor using **`new`** keyword.
- When using **`new`** the **`__ proto__`** is NOT set.

Features of Function Constructors

```
function Person(){
        console.log(this);
        this.university = 'MUM';
        year = '2016';
}

var faculty = new Person(); // Person {university: "MUM"} - no year!


Person.prototype.great = function(){
        return 'Hi ' + this.university;
}

faculty.great(); // "Hi MUM"
```

Why this is awesome? Because we can create thousands of objects from the original function constructor with less memory space. And we can extend the functionality of all objects at runtime by adding methods and properties to the `prototype` property. *(not to mix it up with `__proto__` which is used for inheritance)*

Example with Analysis

```
// By convention we use capital first letter for function constructor
function Course (coursename){
        this.coursename = coursename;
        console.log('Function Constructor Invoked!');
}
Course.prototype.register = function(){
        return 'Register ' + this.coursename;
}
var wap = new Course('WAP'); // Function Constructor Invoked!

console.log(wap); // Course {name: "WAP"}
console.log(wap.__proto__); // Course {} - empty!
console.log(wap instanceof Course); // true
console.log(Course.prototype.register); // function(){ ... }
console.log(wap.register()); // Register WAP
```

Built-in Function Constructors

```
var a = new Number(12);
var b = new String("Hello");
var c = new Date(2016, 03, 01);

// Number.prototype, String.prototype, Date.prototype
// are objects with helper methods
// available because objects were created using new() keyword

a.toString(); // "12"
b.italics(); // "<i>Hello</i>"
c.getMonth(); // 3
```

Creating Objects

- Proper way to create objects is: **`Object.create`**`(object)`
- It sets **`__proto__`** property to original object for inheritance.

```
var person = {
        first: 'Default',
        last: 'Default',
        greet: function() { return 'Hi' + this.first; } //use this in functions
}

var asaad = Object.create(person);
asaad.first = 'Asaad';

console.log(asaad); // {first: 'Asaad', last: 'Default', greet: function()}
asaad.great(); // Hi Asaad
```

Example with Analysis

```
// An Object
var course = {
        coursename: 'Default',
        register: function() {
                return 'Register ' + this.coursename;
                }
}
var mwp = Object.create(course);
mwp.coursename = 'MWP';


console.log(mwp); // Object {coursename: "MWP"}
console.log(mwp.__proto__); // Object {coursename: "Default"}
console.log(course.prototype); // undefined
console.log(mwp.prototype); // undefined
console.log(mwp.register()); // Register MWP
```

Polyfill

```
if (!Object.create) {
        Object.create = function (o) {
                if (arguments.length > 1) {
                        throw new Error('Only accepts the first parameter.');
                }
                function F() {} F.prototype = o;
                return new F();
        };
}
```

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

JavaScript

1. JavaScript is a functional OO language that has a shared global namespace for each page and local scope within functions.
2. Closures and objects are fundamental to JavaScript best coding practices, particularly for promoting encapsulation, layering, and abstractions in code.

---

3. **Transcendental consciousness** is the experience of the most fundamental layer of all existence, pure consciousness, the experience of one’s own Self.
4. **Impulses within the transcendental field**: The many layers of abstraction required for sophisticated JavaScript implementations will be most successful if they arise from a solid basis of thought that is supported by all the laws of nature.
5. **Wholeness moving within itself:** In unity consciousness, one appreciates that all complex systems are ultimately compositions of pure consciousness, one’s own Self.