Maharishi University of Management
Department of Computer Science
CS472 Web Programming

Scope, Closures, and Encapsulation in Javascript

References

http://www.cs.washington.edu/education/courses/cse341/10au/lectures/slides/27-scope-closures.pdf

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

http://www.joezimjs.com/javascript/javascript-closures-and-the-module-pattern/

http://www.jblearning.com/catalog/9780763780609/

Lexical scope in Java

- In Java, every block ({ }) defines a scope.

```
public class Scope {
    public static int x = 10;

    public static void main(String[] args) {
        System.out.println(x);
        if (x > 0) {
            int x = 20;
            System.out.println(x);
        }
        int x = 30;
        System.out.println(x);
    }
}
```

Function scope in JavaScript

- In JavaScript, there are only two scopes:
    - **global scope**: global environment for functions, vars, etc.
    - **function scope**: every function gets its own inner scope

```
var x = 10;                           // foo.js
function main() {
    var x = 40;
    print(x);
    x = 20;
    if (x > 0) {
        x = 30;
        print(x);
    }
    var f = function(x) { print(x); }
    f(50);
}  //
x = 70;
```

Another scope example

```
function f() {
    var a = 1, b = 20, c;
    print(a + " " + b + " " + c);          // 1 20 undefined

    // declares g (but doesn't call immediately!)
    function g() {
        var b = 300, c = 4000;
        print(a + " " + b + " " + c);      // 1 300 4000
        a = a + b + c;
        print(a + " " + b + " " + c);      // 4301 300 4000
    }

    print(a + " " + b + " " + c);          // 1 20 undefined
    g();
    print(a + " " + b + " " + c);          // 4301 20 undefined
}  //run it
```

Lack of block scope

```
for (var i = 0; i < 10; i++) {
    print(i);
}
print(i);  // 10
if (i > 5) {
    var j = 3;
}
print(j);  //run it
```

- any variable declared lives until the end of the function
  - lack of block scope in JS leads to errors for some coders
  - this is a "bad part" of JavaScript (D. Crockford)

`var` VS `let` (ES6)

- `var` scope – nearest function scope
- `let` scope – nearest enclosing block

```
function a() {
 for (var x = 1; x < 10; x++) {
 console.log(x);
}
console.log("x: " + x);
//10
}
```

```
function a() {
 for (let x = 1; x < 10; x++) {
 console.log(x);
 }
 console.log("x: " + x);
//ReferenceError: x is not defined
}
```

- `let` has block scope
- Use `let` inside for loops to prevent leaking to Global Scope

let variables in ES

- From ES6, you can use the *let* keyword to declare a variable. The *let* keyword is similar to the *var* keyword. However, variable declared using the *let* keyword is block-scoped, not function-scoped.
- In the following example, we declare the *tmp* variable within a block surrounding by the curly braces {}. The *tmp* variable only exists inside the block, therefore, any reference to it outside of the block will cause a *ReferenceError*.

let example

```
var foo = 20, bar = 10;

{
    let tmp = foo;
    foo = bar;
    bar = tmp;
}

console.log(tmp); // ReferenceError
```

const in ES6

- The *const* keyword works like the *let* keyword, but the variable that you declare must be initialized immediately with a value, and that value can’t be changed afterward.

```
const CODE = 100;
CODE = 200; // TypeError: CODE is read-only
```

Hoisting

Hoisting is JavaScript's default behavior of moving all declarations to the top of the current function. The following both give the same result:

Example 1

```
x = 5;
elem = document.getElementById("demo");
elem.innerHTML = x;
var x;   // Declare x
```

Example 2

```
var x;   // Declare x
x = 5;
elem = document.getElementById("demo");
elem.innerHTML = x;
```

Hoisting

JavaScript only hoists declarations, not initializations. The following Examples are equivalent:

Example 1

```
var x = 5;   // Initialize x
elem = document.getElementById("demo");
elem.innerHTML = x + " " + y;
var y = 7;   // Initialize y
```

Example 2

```
var x = 5;   // Initialize x
var y;       // Declare y
elem = document.getElementById("demo");
elem.innerHTML = x + " " + y;
y = 7;       // Assign 7 to y
```

Main Point

JavaScript has global scope and local scope within functions when variables are declared with var, and now has block scope with const and let.

Science of Consciousness: The experience of transcending opens our awareness to the expanded vision of unbounded awareness, at the same time that it promotes the ability to focus sharply within any local boundaries.

First-class functions

- Functions can be assigned to variables

```
var myfunc = function(a, x) {
 return a * b;
};
```

- Functions can be passed as parameters

```
function apply(a, b, f) {
 return f(a, b);
}
var x = apply(2, 3, myfunc); // 6
```

- Functions can be return values

```
function getAlert(str) {
 return function() { alert(str); }
}
var whatsUpAlert= getAlert("What's up!");
whatsUpAlert(); // “What’s up!”
```

Javascript functions

- Function *parameters* are the names listed in the function definition.
- Function *arguments* are the real values passed to (and received by) the function.
- JavaScript function definitions do not specify data types for parameters.
- JavaScript functions do not perform type checking on the passed arguments.
- JavaScript functions do not check the number of arguments received.
- If a function is called with missing arguments (less than declared), the missing values are set to: *undefined*

`arguments` Object

JavaScript functions have a built-in object called the **`arguments`** *object. The* **`arguments`** *object contains an array of the arguments used when the function is called (invoked).*

```
function findMax() {
        var i;
        var max = -Infinity;
        for (i = 0; i < arguments.length; i++) {
                if (arguments[i] > max) {
                        max = arguments[i];
                }
        }
        return max;
}

var x = findMax(1, 123, 500, 115, 44, 88); // 500
var x = findMax(5, 32, 24); // 32
```

Arrow functions (ES6)

- Arrow functions are function shorthand using => syntax.
- Syntactically similar to Java 8, lambda expressions

- Two factors influenced the introduction of arrow functions:
  - Shorter functions
  - Non-binding of this (covered later)

Arrow functions (ES6)

Arrow functions can be a shorthand for an anonymous function.

```
(arguments) => { return statement } // general syntax
argument => { return statement } // one parameter
argument => statement // implicit return
() => statement // no input
```

```
function multiply (num1, num2) {
    return num1 * num2;
}
var output = multiply(5, 5);
```

```
() => ({    })
```

```
var multiply = (num1, num2) => num1 * num2;
var output = multiply(5, 5);
```

Default Parameters (ES6)

```
function log(x=10, y=5){
    console.log( x + ", " + y);
}

log(); // 10, 5
log(5); // 5, 5
log(5, 10); // 5, 10
```

Rest Operator (ES6)

- A **Rest** syntax allows us to represent variable number of arguments as an Array.
  - Its like `varargs` in Java and has same syntax.
  - Rest parameters should be the last parameter in a function.

```
function sum(x,y, ...more){
        var total = x + y;
        if(more.length > 0){
                for (var i=0;
i<more.length; i++) {
                        total += more[i];
                }
        }
        console.log(total);
}

sum(4,4); // 8
sum(4,4,4); // 12
```

Calling an inner function

```
function init() {   //function declaration
    var name = "Mozilla";
    function displayName() {
        alert(name);
    }
    displayName();
}
init();
```

Returning an inner function

```
function makeFunc() {
  var name = "Mozilla";   //local to makeFunc
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();  //is the local variable still accessible by myFunc?
```

//another reference and demo on inner function scope

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 given a fixed value when "closed over" by a function's environment.

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

Closures in JS

```
var x = 1;
function f() {
    var y = 2;
    var summ= function() {
        var z = 3;
        print(x + y + z);
    };
    y = 10; return summ;
}
var g = f();
g();   // 1+10+3 is 14  -- run it
```

- inner function closes over free variables as it is declared
  - grabs references to the names, not values (sees updates)

Common closure bug

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

```
> funcs[0]();
5
```

```
> funcs[1]();
5
```

- Closures that bind a loop variable often have this bug.
  - Why do all of the functions return 5?

Common closure bug with fix (ES6)

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

```
//ES6 solution:  let vs var
const funcs = [];
for (let i = 0; i < 5; i++) {
 funcs[i] = function() {
    return i;
 };
}
```

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

25

Practical uses of closures

- A closure lets you associate some data (the environment) 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.
- Situations like this are common on the web.
  - an event handlers is a single function executed in response to an event.
    - e.g., DOM and timer event handlers
  - closures also very useful in Javascript for encapsulation and namespace protection

Function factory with closures

- example of closures being helpful with event handling

```
body { font-family: Helvetica, Arial, sans-serif; font-size: 12px; }
h1 { font-size: 1.5em; }
h2 { font-size: 1.2em; }
```

```
<p>Some paragraph text</p>
  <h1>some heading 1 text</h1>
  <h2>some heading 2 text</h2>
  <a href="#" id="size-12">12</a>
  <a href="#" id="size-14">14</a>
  <a href="#" id="size-16">16</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(16);
document.getElementById('size-16').onclick = makeSizer(20);
```

- http://jsfiddle.net/vnkuZ //jsfiddle link
- why does makeSizer need to return a function?
- what is the free variable and why is it needed?

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 private closures.
- also provide powerful way of managing global namespace,
- Here's how to define some public functions that can access private functions and variables, using closures which is also known as the module pattern:
- “Every real JavaScript programmer should know this if he or she wants to become great” Joe Zim

Module pattern

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

- declares and immediately calls an anonymous function
  - parens around function are a special syntax that means this is a function expression that will be immediately invoked
    - "immediately invoked function"
  - 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 example

```
// old: 3 globals

var count = 0;
function incr(n) {
  count += n;
}
function reset() {
  count = 0;
}
incr(4);  incr(2);
document.write(count);
```

```
// new: 0 globals!
(function() {
    var count = 0;
    function incr(n) {
        count += n;
    }
    function reset() {
        count = 0;
    }
    incr(4);  incr(2);
    document.write (count);
})();  //run it
```

- declare-and-call protects your code and avoids globals
  - avoids common problem with namespace/name collisions

Implied globals

```
name = value;
```

```
function foo() {
    x = 4;
    print(x);
}   // oops, x is still alive now (global)
```

- if you assign a value to a variable without `var`, JS assumes you want a new *global* variable with that name
  - hard to distinguish
  - this is a "bad part" of JavaScript (D.Crockford)

Main Point

2. 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.

Javascript Objects

How about classes and objects?

- small programs are easily written without objects
- JavaScript treats functions as *first-class citizens*
- larger programs become cluttered with disorganized functions
- objects group *related data and behavior*
    - helps manage size and complexity, promotes code reuse
- You have already *used* many types of JavaScript objects
    - Strings, arrays, HTML / XML DOM nodes
    - global DOM objects
    - The jQuery object (following lessons)

Javascript objects

- objects in Javascript are more 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, 'c': 99, 'd': 199,
  'mult': function(a, b) {
      return a * b; }
  };

Common examples of using object literals

```
$.ajax("http://example.com/app.php", {
   'method': "post",          // an object with a field named method (String)
   'timeout': 2000                  // and a field name timeout
});

$("<div>",
  {'css': {                                  // a css field
            'color': red
            },
   'id': 'myid',                       // an id field
   'click': myClickHandler          // and a method called click
});
```

- the parameters in {} passed to jQuery functions and methods are object literals
- object literals are the basis of JSON

Objects that have behavior

```
var name = {
    ...
    methodName: function(parameters) {
        statements;
    }
};

var pt = {
    x: 4,  y: 3,
    distanceFromOrigin: function() {
        return Math.sqrt(this.x * this.x + this.y * this.y);
    }
};

alert(pt.distanceFromOrigin());   // 5
```

- like in Java, objects' methods run "inside" that object
  - inside an object's method, the object refers to itself as this
  - unlike in Java, the this keyword is mandatory in JS

The global object

- technically *no* JavaScript code is "static" in the Java sense
  - *all* code lives inside of some object
  - there is *always* a `this` reference that refers to that object

- all code is executed inside of a **global object**
  - in browsers, it is also called `window`; in Rhino: `global()`
  - global variables/functions you declare become part of it
    - they use the global object as `this` when you call them

- *"JavaScript's global object [...] is far and away the worst part of JavaScript's many bad parts."* -- D. Crockford

Global object and this keyword

```
function printMe() {
    print("I am " + this.name);
}
> var name = “Alfred E. Newman”;
> var teacher = {
           name: “Prof. Tyler Durden”
           department: “CS”
};
> teacher.print = printMe;
> teacher.print();
I am Prof. Tyler Durden
> printMe();
I am Alfred E. Newman
```

for each over object literal

```
var things = {'a': 97, 'b': 98, 'c': 99 };
for (key in things) {
    console.log(key + ', ' + things[key]);
}
```

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

Emulating private methods with closures (module pattern)

```
var counter = (function() {//the parens surrounding the function are JS syntax for immediate evaluation
  var privateCounter = 0;  //private data
  function changeBy(val) {  //private inner function
    privateCounter += val;
  }
  return {
    increment: function() {// three public functions are closures that share the same environment.
      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 */

//additional reference and demo on closures and the module pattern
```

Emulating private methods with closures

- We could store this function in a separate variable and use it to create several counters.

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    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();
counter1.increment();
alert(counter1.value()); /* Alerts 2 */
counter1.decrement();
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

- Could the immediate evaluation syntax be used here also?

PROGRAMMING WITH

JavaScript

Algorithms and Applications for Desktop and Mobile Browsers

John David Dionisio
Ray Toal

Chapter 5 *Functions*

Functions as properties of an object

- Grouping related functions as properties of a single object help organize large programs
- Keep the global namespace clean
  - Important for performance
  - Critical for avoiding name clashes between scripts

```
var Geometry = {
    circleArea: function (radius) {
        return Math.PI * radius * radius; },

    circleCircumference: function (radius) {
          return 2 * Math.PI * radius;    },

    sphereSurfaceArea: function (radius) {
          return 4 * Math.PI * radius * radius;    },

    boxVolume: function (length, width, depth) {
          return length * width * depth;    }
};
```

Use Object.create: A circle datatype

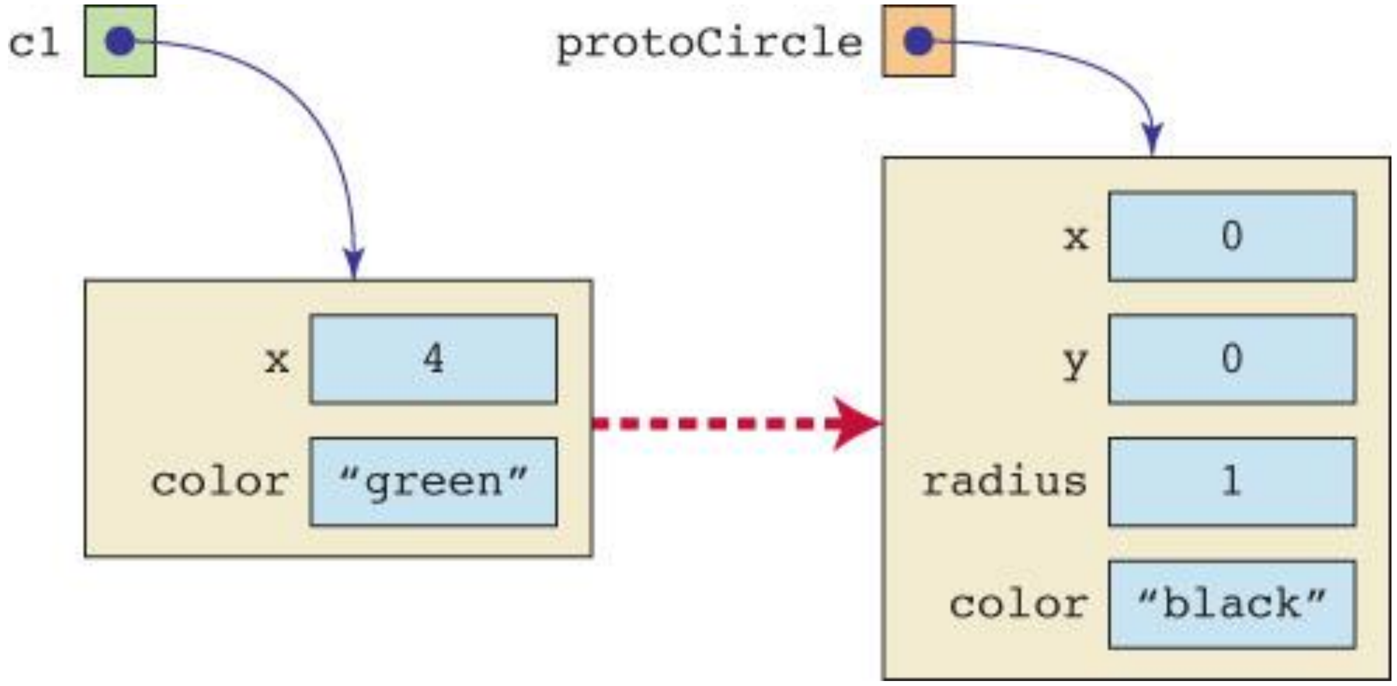
```
/* A prototypical circle, designed to be the prototype for all circles
created with the Circle function below.
*/
var protoCircle = {
   radius: 1,
   area: function () {return Math.PI * this.radius * this.radius;},
   circumference: function () {return 2 * Math.PI * this.radius;}
};

var circle = function (r) {
   var circ = Object.create(protoCircle);  //protoCircle assigned to prototype prop
   circ.radius = r;
   return circ;
};

/* Creates a circle with a given radius. */
  var c = circle(5);
  c.radius => 5
  c.area() => 25pi
  c.circumference() => 10pi
```

Object Prototypes

- Every object has a hidden link to another object, called its *prototype*
- JavaScript looks at prototype if can’t find a property
  - If still not there, looks to prototype's prototype
  - ... and so on until there is no prototype
- Objects therefore have two kinds of properties:
  - *Own* properties
  - *Inherited* properties
- Prototypes are attached to objects at creation time with `Object.create()`

Object Prototype Example 1

```
var protoCircle = {x: 0, y: 0, radius: 1, color: "black"};
var c1 = Object.create(protoCircle);
c1.x = 4;
c1.color = "green"; // Now c1.y === 0 and c1.radius === 1
```



Object Prototype Example 2

Define a prototype guitar and three guitars based on that prototype as follows:

- prototype
  - six-stringed, steel-string, right-handed, acoustic, mahogany.
- first guitar
  - six-stringed, steel-string, left-handed, electric, mahogany, Fender.
- second guitar
  - twelve-stringed, steel-string, right-handed, electric, mahogany, 1953 Les Paul signed by Billie Joe Armstrong.
- third guitar
  - six-string, nylon-string, right-handed, acoustic, 1976 Gibson Explorer Limited Edition of unknown composition, owned by The Edge.

Object Prototype Example 2 (Cont'd)

```
// Prototype: six-stringed, steel-
// string, right-handed, acoustic,
// mahogany
var plainGuitar = {
    strings: 6,
    stringType: "steel",
    hand: "right",
    sound: "acoustic".
    construction: "mahogany"
};


var g1 = Object.create(plainGuitar);
g1.hand = "left";
g1.sound = "electric";
g1.make = "Fender";
```

```
var g2 = Object.create(plainGuitar);
g2.strings = 12;
g2.sound = "electric";
g2.year = 1953;
g2.make = "Les Paul";
g2.signedBy = "Billie Joe Armstrong";


var g3 = Object.create(plainGuitar);
g3.stringType = "nylon";
g3.year = 1976;
g3.make = "Gibson";
g3.model = "Explorer Ltd Edition";
g3.construction = undefined;
g3.owner = "The Edge";
```

Object Prototype Example 3

```
var p = {
  b: 3,
  c: 4
};
var r = Object.create(p);
r.a = 1;
r.b = 2;

console.log(r.a);
/* Is there an 'a' own property on r? Yes, and its value is 1. */

console.log(r.b);
/* Is there a 'b' own property on r? Yes, and its value is 2. The
prototype also has a 'b' property, but it's not visited. This is
called “property shadowing”. */

console.log(r.c);
/* Is there a 'c' own property on r? No, check its prototype. Is
there a 'c' own property on p? Yes, its value is 4. */
```

Keyword *this* in Object Prototypes

```
var o = {
  a: 2,
  m: function(b){
    return this.a + 1;
  }
};

console.log(o.m()); // 3
// When calling o.m in this case, 'this' refers to o

var p = Object.create(o);
// p is an object that inherits from o

p.a = 4; // creates an own property 'a' on p
console.log(p.m()); // 5
// when p.m is called, 'this' refers to p.
// So when p inherits the function m of o,
// 'this.a' means p.a, the own property 'a' of p
```

Creating Objects with “new”

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}

var george = new Person("George Smith", 33, "M");
var ken = new Person("Ken Jones", 39, "M");
```

- *New* operator creates a new object and calls the constructor function to initialize the fields.
- The keyword *this* inside the constructor function points to the newly created object.

Creating Objects with “new”

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
var car1 = new Car("Toyota", "Camry", 1996, george);
var car2 = new Car("Nissan", "Altima", 2016, ken);
```

- Instead of passing a literal string or integer value when creating the new objects, the above statements pass the objects *george* and *ken* as the parameters for the owners.
- To find out the name of the owner of *car2,* access the following property:

```
car2.owner.name
```

Main Point

3. 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**: All objects in the universe are excitations of the field of pure consciousness.