JavaScript – 2 (Functions)

CPEN 400A – Lecture 3

(Loosely based on the book “JavaScript: The Good Parts” by Doug Crockford, O’Reilly Press

Recap: Previous lecture

- In JavaScript, everything is an object
  - Objects are simply hash-tables of key-value pairs
- Objects can be created using either constructor functions or Object.create
  - Possible to support inheritance through prototype
- Reflection is permitted on JavaScript Objects

This lecture

- **Functions in JavaScript: Creation**
- Invoking a function
- Arguments and Exceptions
- Nested functions and closures
- Higher-order functions and Currying

Note about Functions

- Functions are one of the most powerful features in JavaScript, and it is here that JS really shines (for the most part)

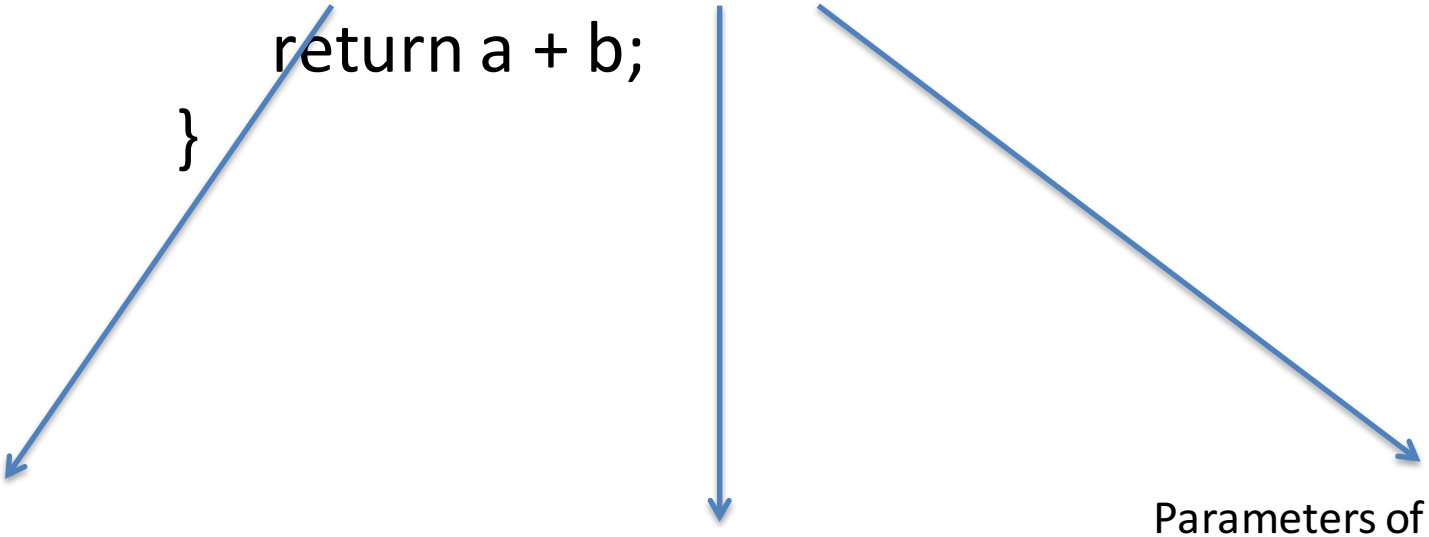
- However, there are some important differences between functions in JS and other imperative languages, such as Java
  - We'll touch upon some of these differences here

Important differences with Java

- In JavaScript, functions are (Data) objects
  - Can be assigned to variables and invoked
  - Can be properties of an object (methods)
  - Can be passed around to other functions
- Functions can be nested inside other functions
  - Can be used to create what are known as closures
- Functions can be called with fewer or more arguments than they take in their parameter lists
  - Can be used to create curried functions

Creating a function

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



Variable to which function is assigned

Function has no name – anonymous. Can specify name.

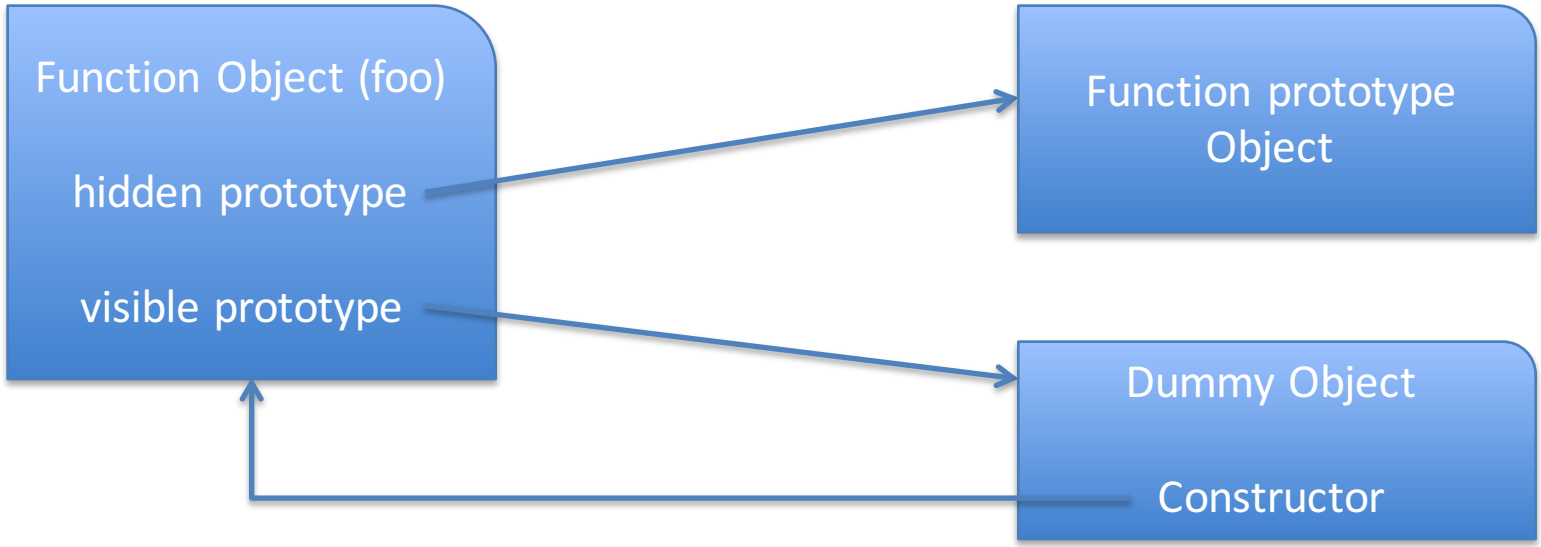
Parameters of the function – set to arguments passed in, undefined if none

Functions are Objects too !

- Every function is an instance of a *Function* object, which is itself derived from *Object*

- A function object has two prototype fields:
  - A *hidden* prototype field to *Function.prototype,* which in turn links to *Object.prototype*
  - A visible prototype field (Function.prototype) which points to an Object whose constructor function points to the function itself !

What's really going on ?



Why is it done in this convoluted way ?

Reason: Constructors

- In JavaScript, Functions can be used as constructors for Object creation (*new operator)*
  - However, JS engine does not know ahead of time which functions are constructors and which aren't
  - For the constructor functions, the (visible) prototype is copied to the new object's prototype
  - New object's prototype's constructor is thus set to the constructor function that created the object

Example

```
function Point( x, y) {
        this.x = x; this.y = y;

};
…
…
var p1 = new Point(2,3);
var p2 = new Point(5,7);
…
console.log(Object.getPrototypeOf(p1) ==
Object.getPrototypeOf(p2));
console.log(Object.getPrototypeOf(p1).constructor);
```

Methods

- Functions can be properties of an Object
  - Analogous to Methods in classical languages
  - Need to explicitly reference *this* in their bodies

```
this.dist = function(point) {

    return Math.sqrt( (this.x – point.x) * (this.x – point.x)
                    + (this.y – point.y) * (this.y – point.y) );
    }
```

NOTE: *this* is bound to the object on which it is invoked

Adding functions to Prototype

- Functions can also be added to the Prototype object of an object
  - These will be applied to all instances of the object
  - Can be overridden by individual objects if needed

```
Point.prototype.toString = function( ) {
        return “(“ + this.x + “ , “ + this.y + “)”;
}
```

This lecture

- Functions in JavaScript: Creation
- **Invoking a function**
- Arguments and Exceptions
- Nested functions and closures
- Higher-order functions and Currying

Invoking Functions

- There are four ways to invoke functions in JS
  - Using function name (for standalone functions)
  - Method calls (for functions in Objects)
  - Constructors (we have seen this earlier)
  - Using Function.apply

- Each of these methods has different bindings of the *this* parameter

Calling a method

- Simply say object.methodName(parameters)

- Example:

```
p1.dist( p2 );
```

this is bound to the object on which it is called. In the example, *this = p1.* This binding occurs at invocation time (late binding).

Calling a standalone function

- If the function is a Standalone one, then the object is called with the *global* context as this
  - Can lead to some strange situations (later)
  - A mistake in the language according to Crockford !

```
var add = function( p1, p2) {
    return new Point(p1.x + p2.x, p1.y + p2.y);
}

add( p1, p2 );
```

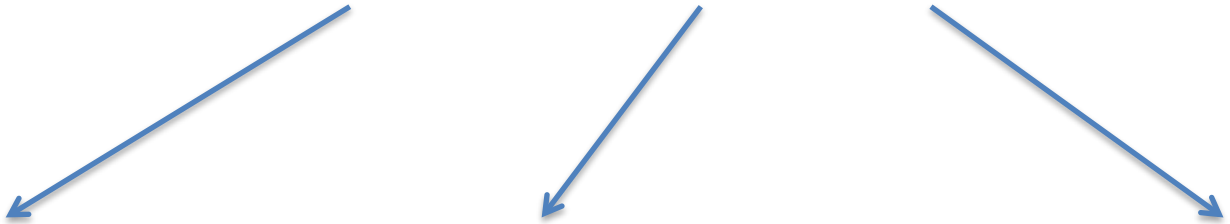
Constructors

- Using the new operator as we've seen
- *this* is set to the new object that was created
  - Automatically returned unless the constructor chooses to return another object (non-primitive)
- Bad things can happen if you forget the 'new' before the call to the constructor (Why ?)

Function.apply

- Most general way to call a function
  - Can set *this to any* arbitrary object in program
  - Can emulate the other three ways of invocation
  - Can also use *call* with the arguments specified

Example: add.apply(null, arguments);



function name to invoke

can be any object, including null

array for passing the arguments

Function.apply example

```
var add2 = function( point1, point2 ) {
        var p = Object.create(this);
        p.x = point1.x + point2.x;
        p.y = point1.y + point2.y;
        return p;
}

var Points = [ p1, p2 ];
var p = add2.apply( Object.getPrototypeOf(p1),  Points);
document.writeln(p);
```

this is bound to the prototype of p1, which is *Point*

Function.call

- Call is similar to apply except that the arguments are specified directly as part of the function parameters rather than in an array
- We used call before for calling the super-class's constructor (for inheritance)

Example:

```
var p = add2.call( Object.getPrototypeOf(p1),  p1, p2);
document.writeln(p);
```

Class Activity

- Emulate the *new* operator through a function *new* using Object.create and Function.apply. Add this function to the 'Point'. This should not duplicate the constructor's code, but invoke it.
- You can access arguments of a function in the array *arguments* from within the function.
- To call this function, you'd write code like:

```
var p1 = Point.new(2, 5);
var p2 = Point.new(3, 7);
```

This lecture

- Functions in JavaScript: Creation
- Invoking a function
- **Arguments and Exceptions**
- Nested functions and closures
- Higher-order functions and Currying

Arguments

- JavaScript does not enforce any rules about function parameters matching their arguments in number (or type for that matter)
- Any additional arguments are simply disregarded (unless function accesses them)
- Fewer arguments mean the remaining parameters are set to undefined

Variadic Functions

- Functions can access their arguments using the *arguments* array
  - Excess parameters are also stored in the array

```
var addAll = function( ) {
        var p = Object.create(this);
        p.x = 0; p.y = 0
        for (var i=0; i<arguments.length; i++) {
                point = arguments[i];
                p.x = p.x + point.x;
                p.y = p.y + point.y;
        }
        return p;
}
```

Return Values

- Functions can return anything they like
  - Objects, including other functions (for closures)
  - Primitive types including null
- If the function returns nothing, it's default return value becomes *undefined*
- The only exception is if it's a constructor
  - Returning object will cause the new object to be lost !

Exceptions

- Functions may also throw exceptions
  - Exception can be any object, but it's customary to have an exception name and an error message
  - Other fields may be added based on context

- Exceptions are caught using try…catch
  - Single catch block for the try
  - Catch can do whatever it wants with the exception, including throwing it again

Exception: Example

```
var addAll = function( ) {
      var p = Object.create(this);
      p.x = 0; p.y = 0
      for (var i=0; i<arguments.length; i++) {
            var  point = arguments[i];
            if ( Object.getPrototypeOf(point) != this )
                  throw { name: TypeError,
                        message: "Object " + point + " is not of type Point “
                  };
            p.x = p.x + point.x;
            p.y = p.y + point.y;
      }
      return p;
}
```

Class Activity

- Modify the *addAll* code to make sure you return the sum so far if the exception is thrown, i.e., sum of elements till the faulty element (you may modify the exception object as you see fit).
- Write code to invoke the addAll function correctly, and to handle the exception appropriately.

This lecture

- Functions in JavaScript: Creation
- Invoking a function
- Arguments and Exceptions
- **Nested functions and closures**
- Higher-order functions and Currying

Nested Functions: Closures

- In JavaScript, functions can nest inside other functions, unlike in languages like Java
- Nested functions can access their enclosing function's properties (this is a good thing)
- However, nested functions cannot access the parent function's *this* and *arguments* (bad)

Closures

- A closure is a nested function that "remembers" the value of it's enclosing function's variables

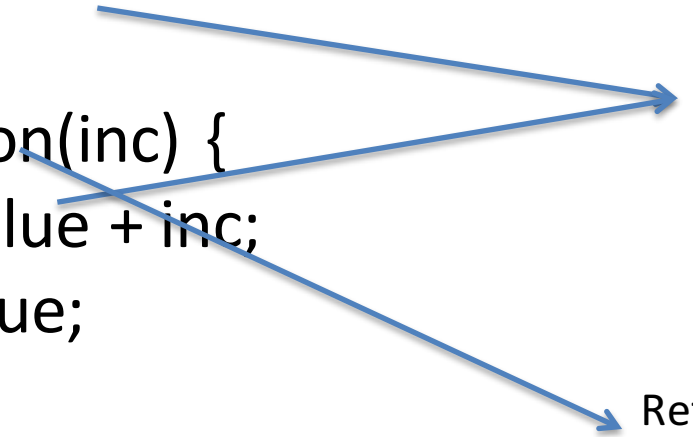
- Can be used for implementing simple, stateful objects
  - Allow variables to be hidden from other objects
  - Can allow objects to be constructed in parts

Closures: Example

```
function Adder(val) {
    var value = val;

    return function(inc) {
        value = value + inc;
        return value;
    }
};

var f = Adder(5);
document.writeln( f(3) );
document.writeln( f(2) );
```



Can access parent function's local variable

Returns a function that needs to be invoked to get it to perform operation

Prints 8

Prints 10

Another Example of Closures

```
function Counter( initial ) {
    var val = initial;
    return {
        increment: function() { val += 1; } ,
        reset: function() { val = initial; }
        get: function() { return val; }
    }
};

var f = Counter(5), g = Counter(10);
f.increment();  f.reset(); f.increment();
g.increment(); g.increment();
console.log( f.get() + " , " + g.get() );
```

Why closures are useful ?

- Allow you to remember state in Web Applications
  - Especially when you have many different handlers construct parts of an object (e.g., AJAX messages)
  - Very useful for callbacks in JavaScript: return the callback function from the parent function
  - Way to emulate private variables (JS has none)

- Closures are extensively used in frameworks such as **jquery** to protect the integrity of internal state

Closures: Referencing Parent Object

- In a closure, what does *this* refer to ?
  - The nested function scope

- But what if you wanted to access the parent function's context (e.g., to invoke a method) ?
  - You no longer get access to parent's *this*
  - Store the parent context in a local variable *that*

Caution: Can lead to high memory consumption

Referencing Parent Object: Example

```
// Implements a closure with multiple counters
function MultiCounter( initial ) {
    var that = this;     // Keep track of the this variable for nested functions
    var val = [];        //  Empty array of counter values
    this.init = function() {
            // Initialize the values of  val from the initial array
            val = [];
            for (var i=0; i<initial.length; i++)
                    val.push( initial[i] );
    };
    this.init();
    return {
        increment: function(i) { val[i] += 1; },
        resetAll: function() { that.init(); },
        getValues: function() { return val; }
    }
};
```

Class Activity- 1

- Assume that you want to maintain an array of N Counter closures (see Slide 32), each starting from a different number 1, 2, 3 etc. Why would the following code (see next slide) not work. Explain why not.

- How would you change the code in the next slide to maintain an array of counters the right way (with distinct values from 1 to n)

Class Activity - 2

```
var MakeCounters = function(n) {
    var counters = [];
    for (var i=0; i<n; i++) {
        var val = i;
        counters[i] = {
            increment: function() { val++; },
            get: function() { return val; },
            reset: function() { val = i; }
        }
    }
    return counters;
}
var m = MakeCounters(10);
for (var i=0; i<10; i++) {
    document.writeln("Counter[ " + i + "] = " + m[i].get());
}
```

Gotchas with Closures

- Remember, the closure stores a link to the variables of the original function, not a copy
  - Any changes to the enclosing variable are reflected in the closure, even after it was created
- Keep the amount of state you want to save in the closure to the minimum necessary state
  - Otherwise, garbage collector cannot release it and you will get memory leaks, and run out of memory

This lecture

- Functions in JavaScript: Creation
- Invoking a function
- Arguments and Exceptions
- Nested functions and closures
- **Higher-Order Functions and Currying**

Higher-order functions

- Passing functions as arguments to other functions to perform some task
  - No need to wrap the function in some weird object as C++ or Java require
  - Function can take any arguments – use apply
- This is very useful for creating generic objects that have ‘plug-and-play’ functionality
- Can also return functions in JS (we just saw this)

Higher Order Function: Example - 1

```
var map = function( array, fn ) {
    // Applies fn to each element of list, returns a new list
    var result = [];
    for (var i = 0; i < array.length; i++) {
        var element = array[i];
        var args = [ element ];
        result.push( fn.apply( null, args) );
    }
    return result;
}

map( [3, 1, 5, 7, 2], function(num) { return num + 10; } );
```

Currying

- Currying is when you want to bind some arguments of a function, so that only the remaining arguments need to be filled in
  - Use *function.bind* to bind some arguments

- Very useful when used in combination with higher-order functions for specifying arguments of functions being passed in

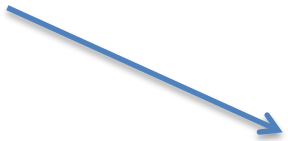
Example of using bind

- Assume that you have a function called foo that takes two arguments

function foo(a, b) { … }

You can bind the first argument to a constant value (or anything else) to return a function goo that takes a single argument as follows.

var goo = foo.bind(null, <value>);



specifies the calling context to bind to

Using currying

- Now you can pass the bound function to the map higher-order function we defined earlier..

```
function add(a, b) { return a + b; }
var add10 = add.bind(null, 10);
// add10 takes a single argument and adds 10 to
// it as the other argument is bound to the value 10
map( [1, 3, 5, 2, 10, 11], add10 );
```

Class Activity - 1

- Write an implementation of *filter* using JavaScript. *filter* takes 2 parameters, an array *arr* and a function *f* that takes a single parameter and returns true or false. It then creates another array with only the elements in *arr* for which *f* returns true.

Class Activity - 2

- Consider a function *lesserThan* that compares two numbers and returns true if the first number is smaller than the second number. Create a curried version of this function to pass to the *filter* function with the first argument set to a user-specified threshold.

- What's the effect of the filter operation here ?

Class Activity: Solution

```
var filter = function(  array, fn ) {
	var result = [];
	for (var i = 0; i < array.length; i++) {
		var element = array[i];
		var args = [ element ];
		if (fn.apply(null,  args) ) result.push(element);
	}
	return result;
};

var lesserThan = function(a,  b) { return (a < b)  ? true:false; };
var greaterThan5 = lesserThan.bind(null,  5);

var a = [ 1, 3, 10, 8, 2, 7, 6 ];
var c = filter( a, greaterThan5);
console.log(c);
```

This lecture

- Functions in JavaScript: Creation
- Invoking a function
- Arguments and Exceptions
- Nested functions and closures
- Higher-Order Functions and Currying