

Lecture 3: JavaScript - Functions

CPEN400A - Building Modern Web Applications - Winter 2021-1

Karthik Pattabiraman

The University of British Columbia
Department of Electrical and Computer Engineering
Vancouver, Canada



Electrical and
Computer
Engineering



Functions in JavaScript: Creation



2

- 1 Functions in JavaScript: Creation
- 2 Invoking a Function
- 3 Arguments and Exceptions
- 4 Nested Functions and Closures
- 5 Higher-Order Functions and Currying

Note about Functions



3

- 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



4

- 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 → Variadic f.
 - Can be used to create curried functions

↙
can pass them
around.



Creating a Standalone Function

\uparrow ref to fx
 No naming. req'd.
 var add = function (a, b) {
 return a + b;
 }
 or
 function foo (a, b)

Variable to which
function is assigned

Function has no
name – anonymous.
Can specify name.

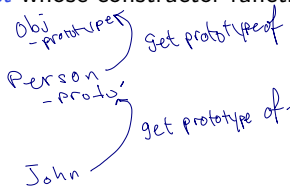
Parameters of the
function – set to
arguments passed in,
undefined if none

Functions are Objects too !



6

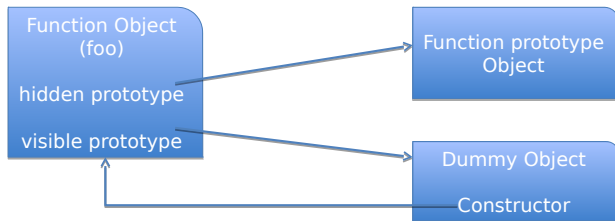
- Every function is an instance of a **Function** object, which is itself derived from **Object**
- A function object has two prototype fields:
 - A hidden prototype field to **Function.prototype**, which in turn links to **Object.prototype**
 - A visible prototype field (**Function.prototype**) which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



What's really going on ?



7



- Why is it done in this convoluted way ?

- JS - interpreter cannot tell if it is constructor.

Reason: Constructors



8

- 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



9

```
1  function Point( x, y ) {  
2      this.x = x; this.y = y;  
3  
4  };  
5  
6  
7  var p1 = new Point(2,3);  
8  var p2 = new Point(5,7);  
9  
10 console.log( Object.getPrototypeOf(p1) == Object.  
    getPrototypeOf(p2)); // true  
11 console.log( Object.getPrototypeOf(p1).constructor); // point
```

Methods



10

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

```
1  this.dist = function(point) {  
2  
3      return Math.sqrt( (this.x - point.x)  
4                          * (this.x - point.x)  
5                          + (this.y - point.y)  
6                          * (this.y - point.y) );  
7  }
```



NOTE

this is bound to the object on which it is invoked

Adding Functions to Prototype



11

- 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

```
1 Point.prototype.toString = function( ) {  
2     return "(" + this.x + " , " + this.y + ")";  
3 }
```

Invoking a Function



12

- 1 Functions in JavaScript: Creation
- 2 Invoking a Function**
- 3 Arguments and Exceptions
- 4 Nested Functions and Closures
- 5 Higher-Order Functions and Currying

Invoking Functions



13

- There are four ways to invoke functions in JS
 - 1 Method calls (for functions in Objects)
 - 2 Standalone function (using function name)
 - 3 Constructors (creating object instances)
 - 4 Using `Function.apply` or `Function.call`
- Each of these methods has different bindings of the `this` parameter

1) Method call (over an object)



14

- `object.methodName(parameters)`
- Example: `p1.dist(p2)`
 ↳ *this =*

NOTE

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

2) Calling a standalone function



15

- 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 !

```
1 var add = function( p1, p2 ) {  
2     return new Point(p1.x + p2.x, p1.y + p2.y);  
3 }  
4  
5 add( p1, p2 );
```

-Don't use "this" in standalone fx

3) Constructors

Avoid returning -

- 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) *Never -* *→ original obj to return*
- Bad things can happen if you forget the **new** before the call to the constructor (Why ?)

↳ call fx as standalone (global context)

4) *Function.apply*



17

- 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
 - **apply** more generic than **call** (i.e., can support variadic arguments). See later for **call**
 - **call** vs **apply**: <https://hangar.runway7.net/javascript/difference-call-apply>

4) *Function.apply* - Example 2



18

```
1  var add2 = function( point1 , point2 ) {  
2      var p = Object.create( this );  
3      p.x = point1.x + point2.x;  
4      p.y = point1.y + point2.y;  
5      return p;  
6  }  
7  
8  var Points = [ p1, p2 ];  
9  var p = add2.apply( Object.getPrototypeOf(p1),  Points );  
10 document.writeln( p );
```

have to be
expanded.

type

↑

some at Object. new p

4) *Function.call*



19

- `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)

```
1 var p = add2.call( Object.getPrototypeOf(p1), p1, p2);  
2 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` class and not through the `Point`'s prototype as it must be called on the class itself and not over a specific instance. 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 (variadic arguments - see later in this presentation).
- To call this function, you'd write code like:

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

Arguments and Exceptions



21

- 1 Functions in JavaScript: Creation
- 2 Invoking a Function
- 3 Arguments and Exceptions**
- 4 Nested Functions and Closures
- 5 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

```
1  var addAll = function( ) {  
2      var p = new Point(0,0);  
3      for (var i=0; i<arguments.length; i++) {  
4          var point = arguments[i];  
5          p.x = p.x + point.x;  
6          p.y = p.y + point.y;  
7      }  
8      return p;  
9  }
```

Return Values



24

- 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



25

- 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



```
1  var addAll = function( ) {  
2    var p = new Point(0,0);  
3    for (var i=0; i<arguments.length; i++) {  
4      var point = arguments[i];  
5      if ( p.x==undefined || p.y==undefined )  
6        throw { name: TypeError,  
7              message: "Object " + point + " is not of type  
8                  Point"  
9      };  
10     p.x = p.x + point.x;  
11     p.y = p.y + point.y;  
12   }  
13   return p;  
14 }
```

Class Activity



27

- 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).

Note

By *return*, we mean that the *caller* will have access to the sum up until the faulty element

- Write code to invoke the `addAll` function correctly, and to handle the exception appropriately.

Nested Functions and Closures



28

1 Functions in JavaScript: Creation

2 Invoking a Function

3 Arguments and Exceptions

4 Nested Functions and Closures

→ event handling in JS

5 Higher-Order Functions and Currying

Nested Functions: Closures



29

- 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**! ↪

Closures

- A closure is a nested function that “remembers” the value of it's enclosing function's variables → every f_x carries environment, state of stk, etc.
- 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

```
1  function Adder(val) {  
2      var value = val;  
3  
4      return function(inc) {  
5          // Returns a function that needs  
6          // to be invoked to get it to  
7          // perform the operation  
8  
9          value = value + inc;  
10         // Can access parent function  
11         // (Adder)'s local variable  
12  
13         return value;  
14     }  
15 };  
16  
17 var f = Adder(5);  
18 document.writeln( f(3) ); // Prints 8  
19 document.writeln( f(2) ); // Prints 10
```

→ copied to value of O.F.

← preserved across call to inner fx

Another Example of Closures



31

```
1 function Counter( initial ) {  
2   var val = initial;  → not instance val / constructor  
3   return {  
4     increment: function() { val += 1; },  
5     reset: function() { val = initial; },  
6     get: function() { return val; }  
7   }  
8 };  
9  
10 var f = Counter(5), g = Counter(10);  
11 f.increment(); f.reset(); f.increment();  
12 g.increment(); g.increment();  
13 console.log( f.get() + " , " + g.get() ); // f=6 g=12
```

← return object of fx.

J.S. hides the
state

Why closures are useful ?



32

- 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



33

- 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 with that: Example



```
1  var MultiCounter = {
2    create: function(initial) {
3
4      var that = this;
5      var val = [];
6      console.log(this);
7      this.init = function() {
8        val = [];
9        for (var i=0; i<initial.length; i++) {
10          val.push( initial[i] );
11        };
12      };
13      this.init();
14      return {
15        increment: function(i) { val[i] += 1; },
16        resetAll: function() { that.init(); },
17        getValues: function() { return val; }
18      };
19    }
20  };
21  var m = MultiCounter.create( [1, 2, 3] );
```

Class Activity- 1



35

```
1  /* 1) Assume that you want to maintain an array of Counter
   closures (see Slide 33), each starting from a different
   number 1, 2, 3 etc. Why would the following code not
   work. Explain why not. */
2
3  var MakeCounters = function(n) {
4    var counters = [];
5    for (var i=0; i<n; i++) {
6      var val = i;
7      counters[i] = {
8        increment: function() { val++; },
9        get: function() { return val; },
10       reset: function() { val = i; }
11     }
12   }
13   return counters;
14 }
15 var m = MakeCounters(10);
16 for (var i=0; i<10; i++) {
17   document.writeln("Counter[ " + i + " ] = " + m[i].get());
18 }
```

Class Activity- 2



```
1  /* 2) How would you change the code to maintain an array of
   counters the right way (with distinct values from 1 to n
   )? (same code below) */
2
3  var MakeCounters = function(n) {
4      var counters = [];
5      for (var i=0; i<n; i++) {
6          var val = i;
7          counters[i] = {
8              increment: function() { val++; },
9              get: function() { return val; },
10             reset: function() { val = i; }
11         }
12     }
13     return counters;
14 }
15 var m = MakeCounters(10);
16 for (var i=0; i<10; i++) {
17     document.writeln("Counter[ " + i + " ] = " + m[i].get());
18 }
```

Gotchas with Closures



37

- 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

Higher-Order Functions and Currying



38

- 1 Functions in JavaScript: Creation
- 2 Invoking a Function
- 3 Arguments and Exceptions
- 4 Nested Functions and Closures
- 5 Higher-Order Functions and Currying

High-Order Functions



39

→ pass f as f

- 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** as seen previously
- This is very useful for creating generic objects that have ‘plug-and-play’ functionality
- Can also return functions in JS, as we’ve just seen

Higher Order Function: Example - 1



40

```
1  var map = function( array , fn ) {  
2    // Applies fn to each element of list, returns a new list  
3    var result = [];  
4    for (var i = 0; i < array.length; i++) {  
5      var element = array[i];  
6      result.push( fn(element) );  
7    }  
8    return result;  
9  }  
10  
11 map( [3, 1, 5, 7, 2], function(num) { return num + 10; } );
```


Currying



41

- Currying: binding 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>);`
 - `null` 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.

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

Value of 'this'.

↳ specialized version of foo.

Summary



1 Functions in JavaScript: Creation

2 Invoking a Function

3 Arguments and Exceptions

4 Nested Functions and Closures

5 Higher-Order Functions and Currying

Class Activity - 1



44

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 ?