Lecture 3: JavaScript - Functions and Objects

Building Modern Web Applications
Vancouver Summer Program 2018 (Package E)

Karthik Pattabiraman, Julien Gascon-Samson

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

Electrical and Computer Engineering

Tuesday, July 24, 2018

Objects in JavaScript

What is an Object in JS ?

- Container of properties, where each property has a name and a value, and is mutable
  - Property names can be any string, including the empty string
  - Property values can be anything except undefined
- What are not objects ?
  - Primitive types such as numbers, booleans, strings
  - null and undefined – these are special types

What is an Object in JS ?

- Container of properties, where each property has a name and a value, and is mutable
  - Property names can be any string, including the empty string
  - Property values can be anything except undefined
- What are not objects ?
  - Primitive types such as numbers, booleans, strings
  - null and undefined – these are special types

What about classes ?

- There are no *classic*, object-oriented classes in JavaScript, as we understand them in languages such as Java
- "What ? How can we have objects without classes ?"
  - Objects use what are known as prototypes
  - An object can inherit the properties of another object using prototype linkage (more later)

Example of Object Creation

```
// Initializing an empty object
var empty_object = {};

// Object with two attributes
var name = {
   firstName: "Karthik",
   lastName: "Pattabiraman";
};
```

NOTE

You don't need a quote around firstName and lastName as they're valid JavaScript identifiers

Retrieving an Object's Property

```
name["firstName"]
// Equivalent to:
name.firstName

name["lastName"]
// Equivalent to:
name.lastName
```

- What if you write `name["middleName"]`?
  - Returns undefined. Later use of this value will result in an "TypeError" exception being thrown

Update of an Object's Property

```
name["firstName"] = "Different firstName";
name.lastName = "Different lastName";
```

- What happens if the property is not present ?
    - It'll get added to the object with the value
- In short, objects behave like hash tables in JS

Objects are passed by REFERENCE !

- In JavaScript, objects are passed by REFRENCE
    - No copies are ever made unless explicitly asked
        - i.e., JSON.parse(JSON.stringify(obj))
    - Changes made in one instance are instantly visible in all instances as it is by reference

Object Constructor and Methods

How to create an object ?

- Define the object type by writing a “Constructor function”
  - By convention, use a capital letter as first letter
  - Use “this” within function to initialize properties
- Call constructor function with the new operator and pass it the values to initialize
  - Forgetting the ‘new’ can have unexpected effects
- ‘new’ operator to create an object of instance ‘Object’, which is a global, unique JavaScript object

Object Creation using New

```
var Person = function(firstName, lastName, gender)
      {
        this.firstName= firstName;
        this.lastName = lastName;
        this.gender = gender;
}
var p = new Person("John", "Smith", "Male");
```

this keyword

- It's a reference to the current object, and is valid only inside the object
- Need to explicitly use this to reference the object's fields and methods
    - Forgetting this means you'll create new local vars
    - Can be stored in ordinary local variables
    - Cannot be modified from within the object

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 (Later)

Object Methods

- Functions that are associated with an object
- Like any other field of the object and invoked as object.methodName()
  - Example: Polygon.draw(10, 100);
  - this is automatically defined inside the method
  - Must be explicitly added to the object

```
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

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).

Functions, arguments and exceptions

Creating a Standalone 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

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 );
```

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 = new Point(0,0);
        for (var i=0; i<arguments.length; i++) {
                var 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 = new Point(0,0);
    for (var i=0; i<arguments.length; i++) {
        var  point = arguments[i];
        if ( p.x==undefined || p.y==undefined )
            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).

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.

Passing functions to other functions

Passing functions to other 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 (will see this later)

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];
        result.push( fn(element) );
    }
    return result;
}

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

Currying

- 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

Currying

- 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.

```
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

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];
       if (fn(element) ) 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);
```