305CDE Lab 3

JavaScript Objects and Functions: Part II

October 2014

Overview

- Objects II
- Functions II

Reminder

Function invocation and `this`.

- The value of `this` inside the scope of the function *depends on how the function was invoked*:
  - method (last week)
  - function (last week)
  - constructor
  - `apply`

Objects II

Constructor Function Invocation (Creates Objects)

- JavaScript actually uses functions to create "class-like objects"
- JS uses the `new` keyword to invoke these special functions

```
function Person(first, last) {
    this.first = first;
    this.last = last;
}
var s = new Person("Simon", "Willison");
```

Constructors 2

`new` is strongly related to `this`:

- creates a brand new empty object
- calls the function specified
- sets `this` to the new object
- returns the new object

Functions that are designed to be called by 'new' are called "constructor functions".

Inheritance With Prototypes

If we can construct “classes”, how do we do inheritance in JS?

- JS is a “prototypal” inheritance language

```
Person.prototype.fullName = function() {
    return this.first + ' ' + this.last;
};
Person.prototype.fullNameReversed = function() {
    return this.last + ', ' + this.first;
};
s.fullName();  // returns "Simon Willison"
```

How does `prototype` work?

- `Person.prototype` is an *object shared by all instances of `Person`*
  - Actually the `prototype` object is a property of all JS objects!
- JavaScript *delegates* to `Person.prototype` if a property is undefined on any `Person` instance
- Anything assigned to `Person.prototype` is available to all instances via the `this` object
  - Note that `var s = new Person("Simon", "Willison");` was executed *before* the `fullName` method was defined
  - Also, the method contained a reference to `this` which picked out the `first` and `last` properties of the object.

Prototypes at Runtime

- JS lets you modify prototypes at runtime
- So you can add extra methods to objects, even built in ones!

```
var s = "Simon";
s.reversed();  // throws TypeError

String.prototype.reversed = function() {
  var r = "";
  for (var i = this.length - 1; i >= 0; i--) {
      r += this[i];
  }
    return r;
};
s.reversed();  // returns "nomiS"
"reverse me".reversed();  // returns "em esrever"
```

Functions II

Function arguments

Every function is passed an array-like object: `arguments`

- Like `this` it is available in all functions
- Holds all of the values passed to the function
- Useful when you want to work with an arbitrary number of arguements

```
function avg() {  // no parameters
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
};
avg(2, 3);  // returns 2.5
avg(2, 3, 4, 5);  // returns 3.5
```

Function Invocation Using apply

- You can manually specify `this` if you need to
- `apply` is a method on function objects taking two parameters:
  1. the value to be bound to `this`
  2. an array of parameters for the function

```
var raceTimes = {
  first: 10.71,
  second: 10.82
};
var newWR = function(current) {
  if (this.first < current) {
    return true;
  }
  return false;
};
newWR.apply(raceTimes, [10.72]);  // returns true
```

Closure

- Inner functions get access to parameters and variables of functions they are inside (except `this` and `arguments`)
- The inner function can "live longer" than its container
- Can be used to maintain state or to protect "private" data:

```
var myObject = (function ( ) {
  var value = 0;  // private data!
  return {
    increment: function (inc) {
      value += inc || 1;
    },
    getValue: function ( ) {
      return value;
    }
  };
}( ));
```

Closure Example

- The previous example creates `myObject` by invoking a function that returns an object literal
- The function defines `value`
- That variable is available to the `increment()` and `getValue()` methods
  - Even when the outer function has completed its execution!
- The `value` is not available to the rest of the program

IIFE Functions

- The previous example uses an *immediately-invoked function expression* (IIFE, pronounced 'IFFY')
- Here is a simpler use:

```
var a = 1;
var b = 2;
(function() {
  var b = 3;
  a += b;
})();
a  // returns 4
b  // returns 2
```

Defining Modules

- IIFEs are useful for defining JS "modules" such as jQuery, YUI, Underscore etc.
- These are self-contained bits of code that can be "imported" into your programs to add functionality
- Here's a little "Counter" module

```
(function (window) {
  function Counter(initialValue) {
    this.value = initialValue;
  }
  Counter.prototype.increment = function(inc) {
    this.value += inc || 1;
  };
  window.myApp = window.myApp || {};
  window.myApp.Counter = Counter;
})(window);
```

Using Modules

- Module code is used just like any other JS
- Access its namespace to use its functionality
- *Any JS program* can include a module JS file, to add its functionality:

```
var myCounter1 = new myApp.Counter(10);
var myCounter2 = new myApp.Counter(0);

myCounter1.value; // returns 10
myCounter2.increment(3);
myCounter2.value;  // returns 3
```