305CDE Lab 3

JavaScript Objects and Functions: Part II

October 2014

Overview

- Objects II
- Functions II

Objects II

Function Invocation and `this`

Remember from last week:

- 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`

Constructor Functions

- JavaScript uses regular functions as classes, but uses `new` to create objects using them

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

- `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

So how do we do inheritance in JS?

- JS is a "prototypal" inheritance language
- Objects inherit properties *from other objects*:

```
Person.prototype.fullName = function() {
    return this.first + ' ' + this.last;
};
Person.prototype.fullNameReversed = function() {
    return this.last + ', ' + this.first;
};
```

- `Person.prototype` is an *object shared by all instances of `Person`*
- 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

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: "2:02:57",
  second: "2:03:45",
  third: "2:03:47"
};

Person.prototype.fullName.apply(raceTimes);
  // returns "2:02:57 2:03:45"
  // because ‘this‘ is ‘raceTimes‘
```

IFFY Functions

- An anonymous function can be used anywhere that you would normally put an expression.
- This allows you to emulate block scope by "hiding" local variables:

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

- This example uses an *immediately-invoked function expression* (IIFE, pronounced 'IFFY')
- Very handy - particularly for defining JS "modules" such as jQuery, YUI, etc.

Constructor Invocation

```
var Quo = function(string) {
  this.status = string;
};

Quo.prototype.get_status = function() {
  return this.status;
};

var myQuo = new Quo("confused");
myQuo.get_status();  // returns "confused"
```

- Keyword `new` used during function invocation
- Creates a new object which includes a link to the function's `prototype` member
- Keyword `this` becomes bound to the new object
- Constructor invocation returns the object

References

Check These For Further Details

Immediately-Invoked Function Expression JavaScript Module Pattern: In-Depth