CS171 - Reading - Lab 6

Excerpts from the book:

Eloquent JavaScript - A Modern Introduction to Programming by Marijn Haverbeke

Functions, parameters and scopes

A function can have multiple parameters or no parameters at all. In the following example, `makeNoise` does not list any parameter names, whereas `power` lists two:

```
var makeNoise = function() {
    console.log("Pling!");
}

makeNoise () ;
// → Pling!

var power = function(base, exponent) {
    var result = 1;

    for(var count=0; count<exponent; count++)
        result *= base ;

    return result ;
}

console.log(power(2, 10));
// → 1024
```

The parameters to a function behave like regular variables, but their initial values are given by the caller of the function, not the code in the function itself. An important property of functions is that the variables created inside of them, including their parameters, are *local* to the function. This means, for example, that the result variable in the `power` example will be newly created every time the function is called, and these separate incarnations do not interfere with each other.

This “localness” of variables applies only to the parameters and to variables declared with the var keyword inside the function body. Variables declared outside of any function are called *global*, because they are visible throughout the program. It is possible to access such variables from inside a function, as long as you haven’t declared a local variable with the same name.

The following code demonstrates this. It defines and calls two functions that both assign a value to the variable `x`. The first one declares the variable as local and thus changes only the local variable. The second does not declare `x` locally, so references to x inside of it refer to the global variable `x` defined at the top of the example.

```
var x = "outside";

var f1 = function() {
    var x = "inside f1";
};

f1();
console.log(x);
// → outside

var f2 = function() {
    x = "inside f2";
};

f2();
console.log(x);
// → inside f2
```

This behavior helps prevent accidental interference between functions. If all variables were shared by the whole program, it'd take a lot of effort to make sure no name is ever used for two different purposes. And if you did reuse a variable name, you might see strange effects from unrelated code messing with the value of your variable. By treating function-local variables as existing only within the function, the language makes it possible to read and understand functions as small universes, without having to worry about all the code at once.

Objects as a programming construct

This story, like most programming stories, starts with the problem of complexity. One philosophy is that complexity can be made manageable by separating it into small compartments that are isolated from each other. These compartments have ended up with the name *objects*. An object is a hard shell that hides the gooey complexity inside it and instead offers us a few knobs and connectors (such as methods) that present an *interface* through which the object is to be used. The idea is that the interface is relatively simple and all the complex things going on *inside* the object can be ignored when working with it.

As an example, you can imagine an object that provides an interface to an area on your screen. It provides a way to draw shapes or text onto this area but hides all the details of how these shapes are converted to the actual pixels that make up the screen. You'd have a set of methods —for example, `drawCircle` —

and those are the only things you need to know in order to use such an object.

Methods

Methods are simply properties that hold function values. This is a simple method:

```
var rabbit = {};
    rabbit.speak = function (line) {
    console.log("The rabbit says '" + line + " '");
};
rabbit.speak ("I'm alive.");
// → The rabbit says "I'm alive."
```

Usually a method needs to do something with the object it was called on. When a function is called as a method — looked up as a property and immediately called, as in `object.method()` — the special variable `this` in its body will point to the object that it was called on.

```
function speak(line) {
    console.log("The " + this.type + " rabbit says '" + line + " '");
}

var whiteRabbit = { type: "white", speak: speak };
var fatRabbit = { type: "fat", speak: speak };

whiteRabbit.speak("Oh my ears and whiskers, " +
" how late it's getting!");
// → The white rabbit says "Oh my ears and whiskers, how late it's getting!"

fatRabbit.speak("I could sure use a carrot right now.");
// → The fat rabbit says "I could sure use a carrot right now."
```

> **The keyword `this`**
>
> In JavaScript, as in most object-oriented programming languages, `this` is a special keyword that is used within methods to refer to the object on which a method is being invoked. The value of this is determined using a simple series of steps:
>
> 1. If the function is invoked using *Function.call* or *Function.apply*, this will be set to the first argument passed to call/apply. If the first argument passed to call/apply is *null* or *undefined*, `this` will refer to the global object (which is the `window` object in Web browsers).

> 2. If the function being invoked was created using *Function.bind*, `this` will be the first argument that was passed to bind at the time the function was created.
> 3. **If the function is being invoked as a method of an object, `this` will refer to that object**
> 4. **Otherwise, the function is being invoked as a standalone function not attached to any object, and `this` will refer to the global object.**
>
> Source: *JavaScript Basics* by Rebecca Murphey

Prototypes

Watch closely:

```
var empty = {};

console.log(empty.toString);
// → function toString ()...{}

console.log(empty.toString());
// → [ object Object ]
```

I just pulled a property out of an empty object. Magic! Well, not really. I have simply been withholding information about the way JavaScript objects work. In addition to their set of properties, almost all objects also have a *prototype*. A prototype is another object that is used as a fallback source of properties. When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on.

So who is the prototype of that empty object? It is the great ancestral prototype, the entity behind almost all objects, *Object.prototype*.

The prototype relations of JavaScript objects form a tree-shaped structure, and at the root of this structure sits *Object.prototype*. It provides a few methods that show up in all objects, such as toString, which converts an object to a string representation.

Constructors

In JavaScript, calling a function with the new keyword in front of it causes it to be treated as a constructor. The constructor will have its this variable bound to a fresh object, and unless it explicitly returns another object value, this new object will be returned from the call.

An object created with `new` is said to be an instance of its constructor.

Here is a simple constructor for rabbits. It is a convention to capitalize the names of constructors so that they are easily distinguished from other functions.

```
function Rabbit(type) {
    this.type = type ;
}

var killerRabbit = new Rabbit("killer");
var blackRabbit = new Rabbit("black");

console.log(blackRabbit.type);
// → black
```

Constructors (in fact, all functions) automatically get a property named `prototype`, which by default holds a plain, empty object that derives from *Object.prototype*. Every instance created with this constructor will have this object as its prototype. So to add a `speak` method to rabbits created with the `Rabbit` constructor, we can simply do this:

```
Rabbit.prototype.speak = function(line) {
    console.log('The ' + this.type + ' rabbit says "' + line + '"') ;
};

blackRabbit.speak("Doom ...");
// → The black rabbit says "Doom ..."
```