CS171 - Reading - Lab 7

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:

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

makeNoise () ;
// → Pling!

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

    for(let 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.

```
let x = "outside";

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

f1();
console.log(x);
// → outside

let 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.

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