Understanding Closures in JavaScript

(Bronx Coffee and Code, 13 April 2017)

Kem Crimmins
email: kem.crimmins@gmail.com
twitter: @kemcrimmins
github: https://github.com/kemcrimmins

Quotations from smart people

"Like objects, closures are a mechanism for containing state. In JavaScript, a closure is created whenever a function accesses a variable defined outside the immediate function scope. It's easy to create closures: Simply define a function inside another function, and expose the inner function, either by returning it, or passing it into another function. **The variables used by the inner function will be available to it, even after the outer function has finished running.**" (Eric Elliott, "The Two Pillars of JavaScript—Pt 2: Functional Programming", emphasis added)

"A closure is the bundling of a function with its lexical environment. Closures are created at function creation time. When a function is defined inside another function, it has access to the variable bindings in the outer function, **even after the outer function exits**." (Eric Elliott, "Why Learn Functional Programming in JavaScript? (Composing Software, emphasis added)

"Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope." (Kyle Simpson, *Scope & Closures* [*You Don't Know JS*])

Demonstrating Closures through Code

Step One: Copy and paste (or type) the following function into the Chrome console.

```
function outer (a) {
     console.log("outer executes" + a);
     function inner(b) {
          console.log("inner executes with " +a+ ", " +b);
     }
     return inner;
}
```

Step Two (proof of concept): Type the below in console .

```
var test = new outer(1);
test(2);
```

Analysis of Closure in Code

```
function outer (a) {
// parameter a is a local variable for function outer
     console.log("outer executes" + a);
     function inner(b) {
     // parameter b is a local variable for function inner, which includes
     // variable a (from function outer) as part of its scope
          console.log("inner executes with " +a+ ", " +b);
     }
     return inner; // function outer exits with this return statement
}

var test = new outer(1);
```

What is the value of test? Its value is what is returned by `outer(1)`**:**

```
// a = 1; is retained as part of inner()'s scope
function inner(b) {
     console.log("inner executes with " +a+ ", " +b);
}
```

```
test(2);
```

What is executed when `test(2)` **is called? The following code is executed:**

```
// a = 1;
function inner(b) {
     // var b = 2;
     console.log("inner executes with " +a+ ", " +b);
}
```

A (perhaps more) Practical Example

Imagine you're making a video game involving cats, and you want to control how fast they move. But you also want to keep your code DRY (Don't Repeat Yourself). The following code might be one way to go:

```
function catMoves(name) {
     console.log(name);
     function moves(speed) {
          console.log(name, speed);
     }
     return moves;
}

var fatFluffy = new catMoves("Fluffy");
fatFluffy(1);

var svelteSam = new catMoves("Sam");
svelteSam(5);
svelteSam(2); // Sam just ate and is slower...
```

Something pretty interesting is going on with the two calls to `svelteSam`. Notice that each execution of `svelteSam` retains access to `name` even though `catMoves` has completed its execution when `var svelteSam` was declared. Also notice that the value of `name` remains the same for each of the calls to `svelteSam`. In other words, `name` is protected from alteration; it is, if you will, a **private variable**.