Closures

In this lesson we will take a look at the interesting phenomenon that is closures.
Let's begin!

WE'LL COVER THE FOLLOWING

- Listing 8-7: Accessing variables defined in another functions' scope.
- Using the `this` object
- Listing 8-8: Unraveling the issue with anonymous functions
- Using variables in anonymous functions and closures
- Listing 8-9: Unexpected behavior of variables in closures

You already learned that functions can return functions. This feature requires that functions have access to variables defined in another function's scope.

Take a look at **Listing 8-7** below. The code defines a `createOp` function that returns another function that uses the `kind` variable within its body. Although `kind` is declared in `createOp`, it is available later in the context of the returned function.

Listing 8-7: Accessing variables defined in another functions' scope.

```
<!DOCTYPE html>
<html>
<head>
  <title>Using closures</title>
  <script>
    var addOp = createOp("add");
    var subtractOp = createOp("subtract");

    console.log(addOp(23, 12));      // 35
    console.log(subtractOp(23, 12)); // 11

    function createOp(kind) {
      return function (a, b) {
        if (kind == "add") {
          return a + b;
        }
        else if (kind == "subtract") {
          return a - b;
        }
      }
    }
  </script>
</head>
<body>
  Listing 8-7: View the console output
</body>
</html>
```

In programming terminology, functions that do not have names are called anonymous functions.

Functions that have access to variables from other function's scopes are called closures.

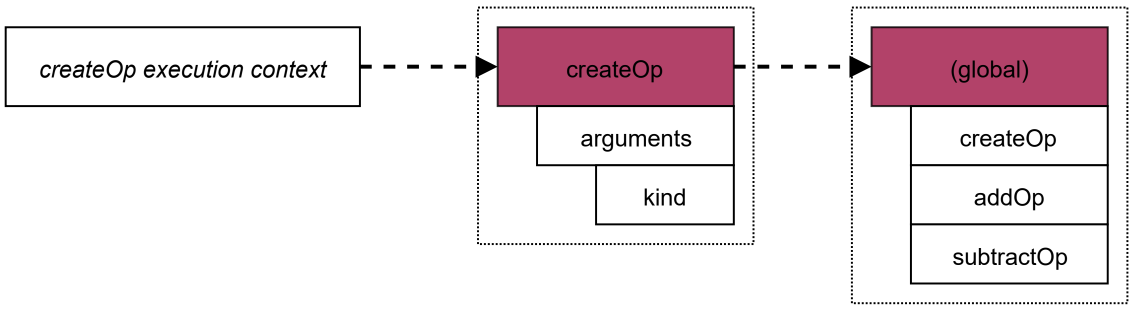
In Listing 8-7, the function returned from `createOp` is an anonymous function, and a closure, too.

To understand how this mechanism works, let's focus on these code lines:

```
var addOp = createOp("add");
// ...
console.log(addOp(23, 12));      // 35
```

When the `createOp` function is called, its execution context looks as shown in the image below.

The context contains two items in its scope chain, the activation object of `createOp` and the global activation object.

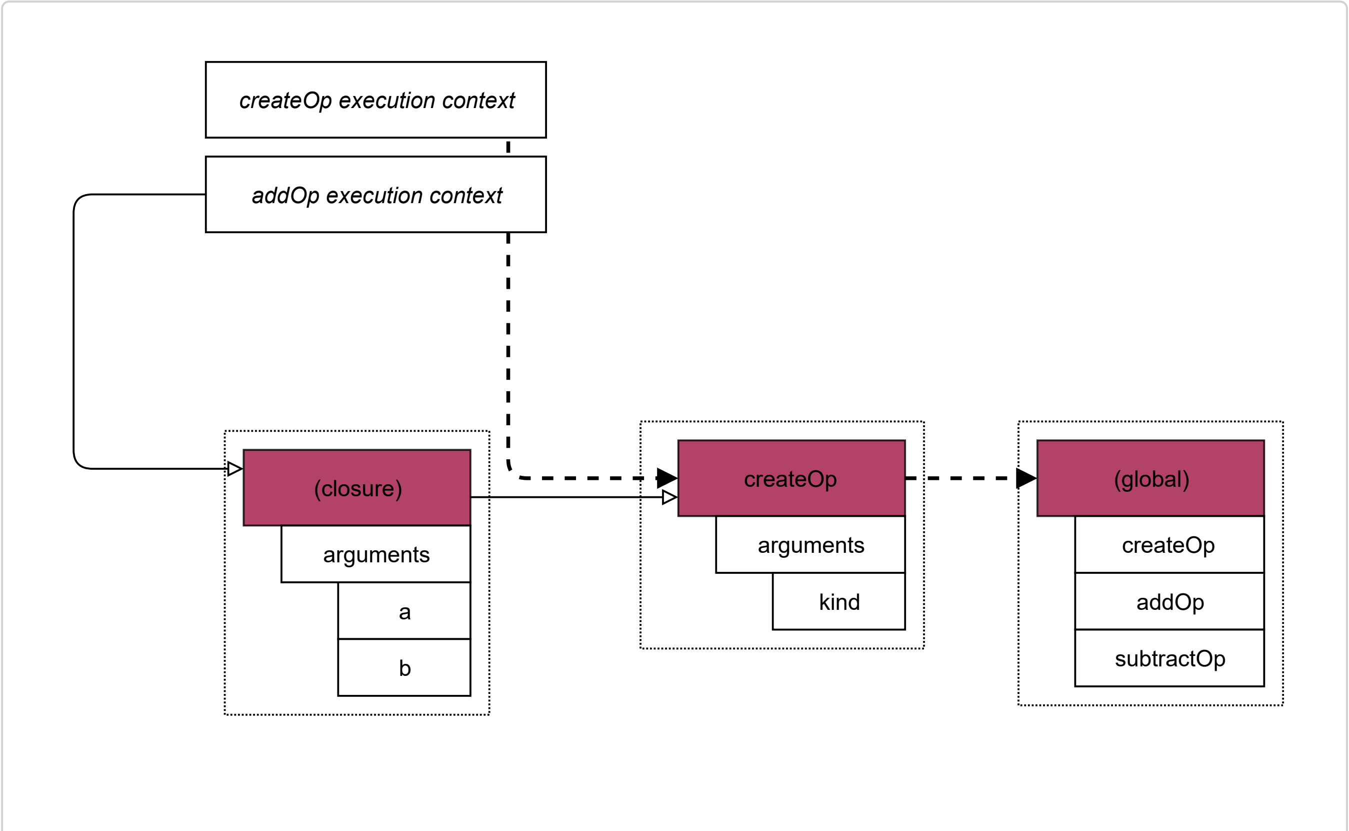


The execution context of createOp

Earlier you learned that whenever a variable is accessed inside a function, the specified name is searched for in the scope chain. Once the function has completed, the local activation object is destroyed, leaving only the global scope in memory.

Closures, on the other hand, behave differently.

A function that is defined inside another function adds the containing function's activation object into its scope chain. This ensures that the internal function has access to the variables of the containing function. So, in `createOp`, the returned function's scope chain includes the activation object for `createOp`. When the `console.log()` operation invokes the anonymous function through `addOp`, the execution context of `addOp` looks like as shown in the image below.



The execution context of addOp

Using the `this` object

Earlier you used the `this` object within constructor functions, such as in this example:

```
var Car = function (manuf, type, regno) {
  this.manufacturer = manuf;
  this.type = type;
  this.regno = regno;
}

// Create a Car
var car = new Car("Honda",
  "FR-V", "ABC-123");
```

The `this` object is bound at run time based on the context in which a function is executed. When called in an object method, this equals the object you invoke the method on. In the constructor function, `this` represents the execution context of the object just being created. When used inside global functions, `this` is equal to the `window` object, which you'll learn about soon.

> 📃**NOTE:** In **strict mode**, this equals undefined when used inside a global function.

There's a little issue with anonymous functions. They are not bound to an object when you invoke them...

So the `this` object points to `window`, or contains `undefined` in strict mode. When you compose a closure, this fact is not really clear.

Listing 8-8 shows an example unraveling this issue.

Listing 8-8: Unraveling the issue with anonymous functions

```
<!DOCTYPE html>
<html>
<head>
  <title>Issue with this</title>
  <script>
    var type = "Mercedes";

    var myCar = {
      type: "BMW",

      getType: function () {
        return this.type;
      },

      getTypeFuncion: function () {
        return function () {
          return this.type;
        }
      }
    };

    console.log(myCar.getType());
    console.log(myCar.getTypeFuncion()());
  </script>
</head>
<body>
  Listing 8-8: View the console output
</body>
</html>
```

The `myCar` object has two functions, the first retrieves the object's type property (getType), and the second returns a function that obtains the value

of the type property. The type property is passed back using the this.type expression in both methods. You expect this code to write "BMW" twice to the console output. However, when running this code, you see this:

console

```
BMW
Mercedes
```

The issue is caused by the this.type expression within getTypeFunction, which retrieves an anonymous function, and this.type is used by the anonymous function, which is a closure, too. When the console.log() operation invokes getTypeFunction(), it is bound to the context of the myCar object.

However, the anonymous function returned by getTypeFunction() is bound to the global context, and so this.type is actually the type variable defined and initialized in the first line of Listing 8-8, and that is why it returns "Mercedes". There is an easy remedy for this issue. Before getTypeFunction() defines the closure, it saves the context, and uses the stored context to access the type property of the object, as this code snippet shows:

```
getTypeFuncion: function () {
  var thisInContext = this;
  return function () {
    return thisInContext.type;
  }
}
```

Using variables in anonymous functions and closures

Sometimes it is not obvious, but variables work in closures unexpectedly.

Listing 8-9 demonstrates this issue.

Listing 8-9: Unexpected behavior of variables in

closures

```
<!DOCTYPE html>
<html>
<head>
  <title>Issues with closure variables</title>
  <script>
    function giveMeFunctions() {
      var functions = [];
      for (var i = 0; i < 3; i++) {
        functions[i] = function () {
          return i * i;
        }
      }
      return functions;
    }

    var myFunctions = giveMeFunctions();
    for (var i = 0; i < myFunctions.length; i++) {
      console.log(i + ": " + myFunctions[i]());
    }
  </script>
</head>
<body>
  Listing 8-9: View the console output
</body>
</html>
```

In this listing, giveMeFunctions() returns an array of functions. The code invokes each function in this array and you expect them to return 0, 1, and 4. However, you see this output:

console

```
0: 9
1: 9
2: 9
```

What's wrong? Each anonymous function retrieves `i*i` where `i` goes from 0 to 2, so something strange must be happening! The code snippet suggests that `i` is evaluated every time the anonymous function is defined, but it does not work this way.

The anonymous function is evaluated when it is invoked and using the execution context it resolves the value of `i`, using the context's scope chain. When you call `giveMeFunctions()`, the `i` variable is set to 3 at the time it returns.

As you iterate through the elements in the returned function array, the `i*i` expression is evaluated using the value of `i`, which happens to be 3 each

time. That is why you see only nines in the output. There is a quick fix for this issue.

Change the definition of `giveMeFunctions`:

```
function giveMeFunctions() {
  var functions = [];
  for (var i = 0; i < 3; i++) {
    functions[i] = function (arg) {
      return function () {
        return arg * arg;
      }
    }(i);
  }
  return functions;
}
```

This definition defines a function with a simple arg that retrieves another function. This internal function captures the arg variable and returns arg's square value. The outer function is immediately called with i passed as its argument due to the (i) appended to the end of the definition. As a result, the three internal anonymous function is bound with three different execution contexts where arg is 0, 1, and 2. They provide the result you expect:

console

```
0: 0
1: 1
2: 4
```

As you can see, JavaScript functions provide a number of great features. Combining them with the versatility of JavaScript objects makes the language very powerful.

In the *next lesson*, we will study usage of objects in JavaScript in more detail.