

Closures

A *closure* is the combination of a function and the lexical environment within which that function was declared.

Lexical scoping

Consider the following:

```
1  function init() {  
2      var name = 'Mozilla'; // name is a local variable created by init  
3      function displayName() { // displayName() is the inner function, a closure  
4          alert(name); // use variable declared in the parent function  
5      }  
6      displayName();  
7  }  
8  init();
```

`init()` creates a local variable called `name` and a function called `displayName()`. The `displayName()` function is an inner function that is defined inside `init()` and is only available within the body of the `init()` function. The `displayName()` function has no local variables of its own. However, because inner functions have access to the variables of outer functions, `displayName()` can access the variable `name` declared in the parent function, `init()`.

JavaScript Result

[Edit in JSFiddle](#)

```
function init() {  
    var name = "Mozilla"; // name is a local variable created by init  
    function displayName() { // displayName() is the inner function, a closure  
        alert (name); // displayName() uses variable declared in the parent function  
    }  
    displayName();  
}
```

🔗 [Run](#) the code and notice that the `alert()` statement within the `displayName()` function successfully displays the value of the `name` variable, which is declared in its parent function. This is an example of *lexical scoping*, which describes how a parser resolves variable names when functions are nested. The word "lexical" refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

Closure

Now consider the following example:

```
1  function makeFunc() {  
2      var name = 'Mozilla';  
3      function displayName() {  
4          alert(name);  
5      }  
6      return displayName;  
7  }  
8  
9  var myFunc = makeFunc();  
10 myFunc();
```

Running this code has exactly the same effect as the previous example of the `init()` function above; what's different — and interesting — is that the `displayName()` inner function is returned from the outer function before being executed.

At first glance, it may seem unintuitive that this code still works. In some programming languages, the local variables within a function exist only for the duration of that function's execution. Once `makeFunc()` has finished executing, you might expect that the `name` variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time that the closure was created. In this case, `myFunc` is a reference to the instance of the function `displayName` created when `makeFunc` is run. The instance of `displayName` maintains a reference to its lexical environment, within which the variable `name` exists. For this reason, when `myFunc` is invoked, the variable `name` remains available for use and "Mozilla" is passed to `alert`.

Here's a slightly more interesting example — a `makeAdder` function:

```
1  function makeAdder(x) {  
2      return function(y) {  
3          return x + y;  
4      };  
5  }  
6  
7  var add5 = makeAdder(5);  
8  var add10 = makeAdder(10);  
9  
10 console.log(add5(2)); // 7  
11 console.log(add10(2)); // 12
```

In this example, we have defined a function `makeAdder(x)`, which takes a single argument, `x`, and returns a new function. The function it returns takes a single argument, `y`, and returns the sum of `x` and `y`.

In essence, `makeAdder` is a function factory — it creates functions which can add a specific value to their argument. In the above example we use our function factory to create two new functions — one that adds 5 to its argument, and one that adds 10.

`add5` and `add10` are both closures. They share the same function body definition, but store different lexical environments. In `add5`'s lexical environment, `x` is 5, while in the lexical environment for `add10`, `x` is 10.

Practical closures

Closures are useful because they let you associate some data (the lexical environment) with a function that operates on that data. This has obvious parallels to object oriented programming, where objects allow us to associate some data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code we write in front-end JavaScript is event-based — we define some behavior, then attach it to an event that is triggered by the user (such as a click or a keypress). Our code is generally attached as a callback: a single function which is executed in response to the event.

For instance, suppose we wish to add some buttons to a page that adjust the text size. One way of doing this is to specify the font-size of the body element in pixels, then set the size of the

other elements on the page (such as headers) using the relative em unit:

```
1  body {
2    font-family: Helvetica, Arial, sans-serif;
3    font-size: 12px;
4  }
5
6  h1 {
7    font-size: 1.5em;
8  }
9
10 h2 {
11   font-size: 1.2em;
12 }
```

Our interactive text size buttons can change the font-size property of the body element, and the adjustments will be picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

```
1  function makeSizer(size) {
2    return function() {
3      document.body.style.fontSize = size + 'px';
4    };
5  }
6
7  var size12 = makeSizer(12);
8  var size14 = makeSizer(14);
9  var size16 = makeSizer(16);
```

size12, size14, and size16 are now functions which will resize the body text to 12, 14, and 16 pixels, respectively. We can attach them to buttons (in this case links) as follows:

```
1  document.getElementById('size-12').onclick = size12;
2  document.getElementById('size-14').onclick = size14;
3  document.getElementById('size-16').onclick = size16;
```

```
1  <a href="#" id="size-12">12</a>
2  <a href="#" id="size-14">14</a>
```

3 | `16`

JavaScript HTML CSS Result

Edit in JSFiddle

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
```

Emulating private methods with closures

Languages such as Java provide the ability to declare methods private, meaning that they can only be called by other methods in the same class.

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code: they also provide a powerful way of managing your global namespace, keeping non-essential methods from cluttering up the public interface to your code.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Using closures in this way is known as the [module pattern](#):

```
1  var counter = (function() {
2    var privateCounter = 0;
3    function changeBy(val) {
4      privateCounter += val;
5    }
6    return {
7      increment: function() {
8        changeBy(1);
9      },
10     decrement: function() {
11       changeBy(-1);
12     },
13     value: function() {
14       return privateCounter;
15     }
16   };
17 })();
```

```
18
19 console.log(counter.value()); // logs 0
20 counter.increment();
21 counter.increment();
22 console.log(counter.value()); // logs 2
23 counter.decrement();
24 console.log(counter.value()); // logs 1
```

In previous examples, each closure has had its own lexical environment. Here, though, we create a single lexical environment that is shared by three functions: `counter.increment`, `counter.decrement`, and `counter.value`.

The shared lexical environment is created in the body of an anonymous function, which is executed as soon as it has been defined. The lexical environment contains two private items: a variable called `privateCounter` and a function called `changeBy`. Neither of these private items can be accessed directly from outside the anonymous function. Instead, they must be accessed by the three public functions that are returned from the anonymous wrapper.

Those three public functions are closures that share the same environment. Thanks to JavaScript's lexical scoping, they each have access to the `privateCounter` variable and `changeBy` function.

You'll notice we're defining an anonymous function that creates a counter, and then we call it immediately and assign the result to the `counter` variable. We could store this function in a separate variable `makeCounter` and use it to create several counters.

```
1  var makeCounter = function() {
2    var privateCounter = 0;
3    function changeBy(val) {
4      privateCounter += val;
5    }
6    return {
7      increment: function() {
8        changeBy(1);
9      },
10     decrement: function() {
11       changeBy(-1);
12     },
13     value: function() {
14       return privateCounter;
15     }
16   }
```

```

17  };
18
19  var counter1 = makeCounter();
20  var counter2 = makeCounter();
21  alert(counter1.value()); /* Alerts 0 */
22  counter1.increment();
23  counter1.increment();
24  alert(counter1.value()); /* Alerts 2 */
25  counter1.decrement();
26  alert(counter1.value()); /* Alerts 1 */
27  alert(counter2.value()); /* Alerts 0 */

```

Notice how each of the two counters, counter1 and counter2, maintains its independence from the other. Each closure references a different version of the privateCounter variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable; however changes to the variable value in one closure do not affect the value in the other closure.

Using closures in this way provides a number of benefits that are normally associated with object oriented programming -- in particular, data hiding and encapsulation.

Creating closures in loops: A common mistake

Prior to the introduction of the `let` keyword in ECMAScript 2015, a common problem with closures occurred when they were created inside a loop. Consider the following example:

```

1  <p id="help">Helpful notes will appear here</p>
2  <p>E-mail: <input type="text" id="email" name="email"></p>
3  <p>Name: <input type="text" id="name" name="name"></p>
4  <p>Age: <input type="text" id="age" name="age"></p>

```



```

1  function showHelp(help) {
2      document.getElementById('help').innerHTML = help;
3  }
4
5  function setupHelp() {
6      var helpText = [
7          {'id': 'email', 'help': 'Your e-mail address'},
8          {'id': 'name', 'help': 'Your full name'},
9          {'id': 'age', 'help': 'Your age (you must be over 16)'}
10     ];

```

```

11
12     for (var i = 0; i < helpText.length; i++) {
13         var item = helpText[i];
14         document.getElementById(item.id).onfocus = function() {
15             showHelp(item.help);
16         }
17     }
18 }
19
20 setupHelp();

```

JavaScript HTML CSS Result

[Edit in JSFiddle](#)

```

function showHelp(help) {
    document.getElementById('help').innerHTML = help;
}

function setupHelp() {
    var helpText = [
        { 'id': 'email', 'help': 'Your e-mail address' },

```

The `helpText` array defines three helpful hints, each associated with the ID of an input field in the document. The loop cycles through these definitions, hooking up an `onfocus` event to each one that shows the associated help method.

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to `onfocus` are closures; they consist of the function definition and the captured environment from the `setupHelp` function's scope. Three closures have been created by the loop, but each one shares the same single lexical environment, which has a variable with changing values (`item.help`). The value of `item.help` is determined when the `onfocus` callbacks are executed. Because the loop has already run its course by that time, the `item` variable object (shared by all three closures) has been left pointing to the last entry in the `helpText` list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier:

```

1 function showHelp(help) {
2     document.getElementById('help').innerHTML = help;
3 }

```



```

4
5 function makeHelpCallback(help) {
6     return function() {
7         showHelp(help);
8     };
9 }
10
11 function setupHelp() {
12     var helpText = [
13         {'id': 'email', 'help': 'Your e-mail address'},
14         {'id': 'name', 'help': 'Your full name'},
15         {'id': 'age', 'help': 'Your age (you must be over 16)'}
16     ];
17
18     for (var i = 0; i < helpText.length; i++) {
19         var item = helpText[i];
20         document.getElementById(item.id).onfocus = makeHelpCallback(item.help)
21     }
22 }
23
24 setupHelp();

```

JavaScript HTML CSS Result

[Edit in JSFiddle](#)

```

function showHelp(help) {
    document.getElementById('help').innerHTML = help;
}

function makeHelpCallback(help) {
    return function() {
        showHelp(help);
    };
}

function setupHelp() {
    var helpText = [

```

This works as expected. Rather than the callbacks all sharing a single lexical environment, the `makeHelpCallback` function creates *a new lexical environment* for each callback, in which `help` refers to the corresponding string from the `helpText` array.

One other way to write the above using anonymous closures is:

```

1 function showHelp(help) {
2     document.getElementById('help').innerHTML = help;
3 }
4
5 function setupHelp() {
6     var helpText = [
7         {'id': 'email', 'help': 'Your e-mail address'},
8         {'id': 'name', 'help': 'Your full name'},
9         {'id': 'age', 'help': 'Your age (you must be over 16)'}
10    ];
11
12    for (var i = 0; i < helpText.length; i++) {
13        (function() {
14            var item = helpText[i];
15            document.getElementById(item.id).onfocus = function() {
16                showHelp(item.help);
17            }
18        })(); // Immediate event listener attachment with the current value of
19    }
20 }
21
22 setupHelp();

```

If you don't want to use more closures, you can use the **let** keyword introduced in ES2015 :

```

1 function showHelp(help) {
2     document.getElementById('help').innerHTML = help;
3 }
4
5 function setupHelp() {
6     var helpText = [
7         {'id': 'email', 'help': 'Your e-mail address'},
8         {'id': 'name', 'help': 'Your full name'},
9         {'id': 'age', 'help': 'Your age (you must be over 16)'}
10    ];
11
12    for (var i = 0; i < helpText.length; i++) {
13        let item = helpText[i];
14        document.getElementById(item.id).onfocus = function() {
15            showHelp(item.help);
16        }
17    }
18 }

```

```
17     }  
18   }  
19  
20   setupHelp();
```

This example uses `let` instead of `var`, so every closure binds the block-scoped variable, meaning that no additional closures are required.

Performance considerations

It is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

Consider the following case:

```
1  function MyObject(name, message) {  
2    this.name = name.toString();  
3    this.message = message.toString();  
4    this.getName = function() {  
5      return this.name;  
6    };  
7  
8    this.getMessage = function() {  
9      return this.message;  
10   };  
11 }
```

Because the previous code does not take advantage of the benefits of closures, we could instead rewrite it as follows:

```
1  function MyObject(name, message) {  
2    this.name = name.toString();  
3    this.message = message.toString();  
4  }  
5  MyObject.prototype = {  
6    getName: function() {
```

```
7     return this.name;
8 },
9 getMessage: function() {
10     return this.message;
11 }
12 };
```

However, redefining the prototype is not recommended. The following example instead appends to the existing prototype:

```
1 function MyObject(name, message) {
2     this.name = name.toString();
3     this.message = message.toString();
4 }
5 MyObject.prototype.getName = function() {
6     return this.name;
7 };
8 MyObject.prototype.getMessage = function() {
9     return this.message;
10 };
```

In the two previous examples, the inherited prototype can be shared by all objects and the method definitions need not occur at every object creation. See [Details of the Object Model](#) for more.

