



Closures

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

Lexical scoping

Consider the following example code:

```
function init() {  
  var name = 'Mozilla'; // name is a local variable created by init  
  function displayName() { // displayName() is the inner function, a closure  
    alert(name); // use variable declared in the parent function  
  }  
  displayName();  
}  
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 available only within the body of the `init()` function. Note that the `displayName()` function has no local variables of its own. However, since inner functions have access to the variables of outer functions, `displayName()` can access the variable `name` declared in the parent function, `init()`.

Run the code using [this JSFiddle link](#) 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

Consider the following code example:

```
    return displayName;
}

var myFunc = makeFunc();
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 might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once `makeFunc()` finishes 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 the closure was created. In this case, `myFunc` is a reference to the instance of the function `displayName` that is 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:

```
function makeAdder(x) {
    return function(y) {
        return x + y;
    };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);
```



value to their argument. In the above example, the function factory creates two new functions—one that adds five 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 data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow you to associate 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 written in front-end JavaScript is event-based. You define some behavior, and then attach it to an event that is triggered by the user (such as a click or a keypress). The code is attached as a callback (a single function that is executed in response to the event).

For instance, suppose we want to add buttons to a page to adjust the text size. One way of doing this is to specify the font-size of the `body` element (in pixels), and then set the size of the other elements on the page (such as headers) using the relative `em` unit:

```
body {  
  font-family: Helvetica, Arial, sans-serif;  
  font-size: 12px;  
}
```



```
function makeSizer(size) {  
  return function() {  
    document.body.style.fontSize = size + 'px';  
  };  
}  
  
var size12 = makeSizer(12);  
var size14 = makeSizer(14);  
var size16 = makeSizer(16);
```



size12, size14, and size16 are now functions that resize the body text to 12, 14, and 16 pixels, respectively. You can attach them to buttons (in this case hyperlinks) as demonstrated in the following code example.

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



```
<a href="#" id="size-12">12</a>  
<a href="#" id="size-14">14</a>  
<a href="#" id="size-16">16</a>
```



Run the code using [JSFiddle](#).

Emulating private methods with closures

Languages such as Java allow you to declare methods as private, meaning that they can be

```
        changeBy(1);
    },

    decrement: function() {
        changeBy(-1);
    },

    value: function() {
        return privateCounter;
    }
};
})();

console.log(counter.value()); // 0.

counter.increment();
counter.increment();
console.log(counter.value()); // 2.

counter.decrement();
console.log(counter.value()); // 1.
```

In previous examples, each closure had its own lexical environment. Here though, there is a single lexical environment that is shared by the three functions: `counter.increment`,

```
    decrement: function() {
      changeBy(-1);
    },

    value: function() {
      return privateCounter;
    }
  }
};

var counter1 = makeCounter();
var counter2 = makeCounter();

alert(counter1.value()); // 0.

counter1.increment();
counter1.increment();
alert(counter1.value()); // 2.
```

```
// global scope
var e = 10;
function sum(a){
  return function(b){
    return function(c){
      // outer functions scope
      return function(d){
        // local scope
        return a + b + c + d + e;
      }
    }
  }
}
```




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



```
function showHelp(help) {
  document.getElementById('help').textContent = help;
```



described earlier:

```
function showHelp(help) {
```





