Closures

Closures are functions whose inner functions refer to independent (free) variables (variables that are used locally, but defined in an enclosing scope). In other words, the functions defined in the closure 'remember' the environment in which they were created.

Lexical scoping

Consider the following:

```
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 `name` and then a function called `displayName()`. `displayName()` is an inner function that is defined inside `init()` and is only available within the body of that function. `displayName()` has no local variables of its own, however it has access to the variables of outer functions and so can use the variable `name` declared in the parent function.

Run the code and see that this works. This is an example of *lexical scoping*: in JavaScript, the scope of a variable is defined by its location within the source code (it is apparent *lexically*) and nested functions have access to variables declared in their outer scope.

Closure

Now consider the following example:

```
function makeFunc() {
  var name = "Mozilla";
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

If you run this code it will have exactly the same effect as the previous `init()` example: the string "Mozilla" will be displayed in a JavaScript alert box. What's different — and interesting — is that the `displayName()` inner function was returned from the outer function before being executed.

That the code still works may seem unintuitive. Normally, the local variables within a function only exist for the duration of that function's execution. Once `makeFunc()` has finished executing, it is reasonable to expect that the name variable will no longer be accessible. Since the code still works as expected, this is obviously not the case.

The solution to this puzzle is that `myFunc` has become a *closure*. A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created. In this case, `myFunc` is a closure that incorporates both the `displayName` function and the "Mozilla" string that existed when the closure was created.

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);
```

```
console.log(add5(2));  // 7
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 environments. In `add5`'s environment, `x` is 5. As far as `add10` is concerned, `x` is 10.

Practical closures

That's the theory out of the way — but are closures actually useful? Let's consider their practical implications. A closure lets you associate some data (the 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 web 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.

Here's a practical example: 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:

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

h1 {
  font-size: 1.5em;
}

h2 {
  font-size: 1.2em;
}
```

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:

```
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 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:

```
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>
```

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.

Here's how to define some public functions that can access private functions and variables, using closures which is also known as the module pattern:

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value()); // logs 0
counter.increment();
counter.increment();
console.log(counter.value()); // logs 2
counter.decrement();
console.log(counter.value()); // logs 1
```

There's a lot going on here. In previous examples each closure has had its own environment; here we create a single environment which is shared by three functions: `counter.increment`, `counter.decrement`, and `counter.value`.

The shared environment is created in the body of an anonymous function, which is executed as soon as it has been defined. The 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.

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
```

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

var counter1 = makeCounter();
var counter2 = makeCounter();
alert(counter1.value()); /* Alerts 0 */
counter1.increment();
counter1.increment();
alert(counter1.value()); /* Alerts 2 */
counter1.decrement();
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

Notice how each of the two counters maintains its independence from the other. Its environment during the call of the `makeCounter()` function is different each time. The closure variable `privateCounter` contains a different instance each time.

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 6, a common problem with closures occurred when they were created inside a loop. Consider the following example:

```
<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').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
      showHelp(item.help);
    }
  }
}

setupHelp();
```

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, but each one shares the same single environment. By the time the `onfocus` callbacks are executed, the loop has run its course and the item variable (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 on:

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

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

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
  }
}

setupHelp();
```

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

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 impractical but demonstrative case:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function() {
    return this.name;
  };

  this.getMessage = function() {
    return this.message;
  };
}
```

The previous code does not take advantage of the benefits of closures and thus could instead be formulated:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype = {
  getName: function() {
    return this.name;
  },
  getMessage: function() {
    return this.message;
  }
};
```

However, redefining the prototype is not recommended, so the following example is even better because it appends to the existing prototype:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function() {
  return this.name;
};
MyObject.prototype.getMessage = function() {
  return this.message;
};
```

The above code can also be written in a cleaner way with the same result:

```
function MyObject(name, message) {
    this.name = name.toString();
    this.message = message.toString();
}
(function() {
    this.getName = function() {
        return this.name;
    };
    this.getMessage = function() {
        return this.message;
    };
}).call(MyObject.prototype);
```

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 details.