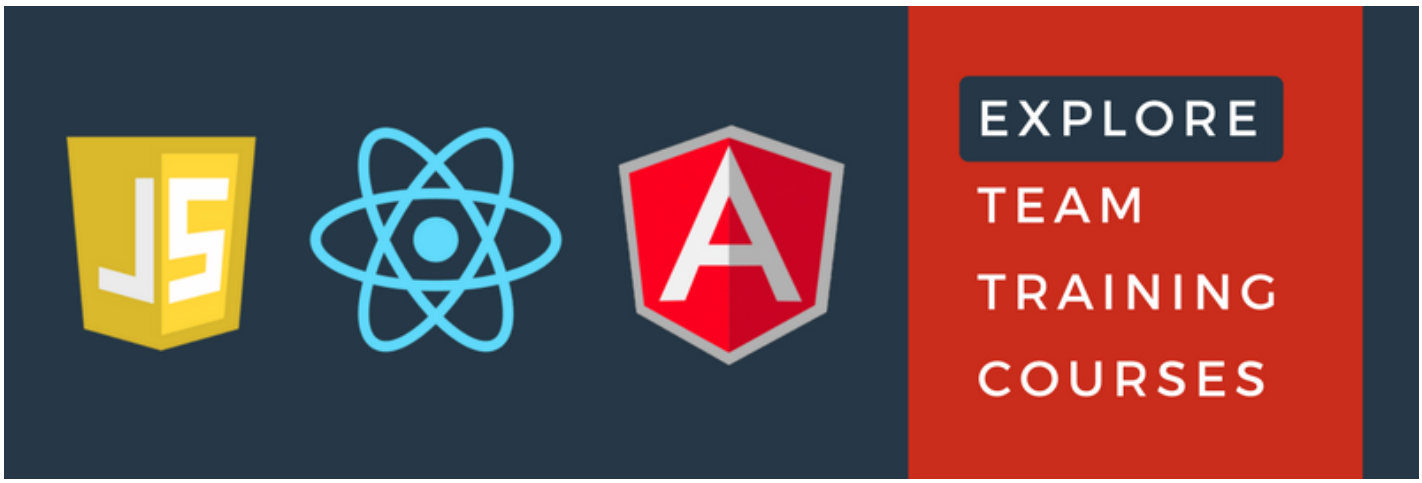
Quick and Dirty JavaScript Closures



Understanding closures is crucial to being a successful javascript developer. The purpose of this post is to provide an introduction to closures, as well as some examples of what you can do with them. This post assumes you have an understanding of how variable scope works in javascript.

Put simply, a closure is a nested function that contains references to the data contained in the same scope as the function. In other words, a closure is a function that is bound to it's environment. Consider the following example:

```
1  var xTen = function() {
2
3  var mult = 10;
4
5  return function(y) {
6
7  console.log(mult*y);
8
9  ;}
10
```

```
11 }();
12
13 xTen(2); // 20
14
15 xTen(5.5); // 55
```

'codepen.io/qualitydixon/pen/zrLLqQ?editors=0011

An anonymous function is defined and invoked and the return value is set to the variable xTen. The return value is a function that logs the product of mult (declared in the scope of the anonymous function) and y (a parameter of the nested function). Then xTen is called twice, each time being passed one parameter, and the product is written to the console.

The interesting thing is that xTen has "remembered" mult. Why is this significant? Because usually locally scoped variables only exist while their containing function is being executed. However, when you define an inner function, the data of the outer function is preserved. Thus the nested function can retain knowledge of its "surroundings" (the data established in the parent scope).

Look at this modified version of the xTen example:

```
1  function makeMult(mult) {
2
3   return function(y) {
4
5       console.log(mult*y);
6
7   }
8
9  };
10
11 var xTen = makeMult(10);
12
13 var xHundred = makeMult(100);
14
15 xTen(55); // 550
16
17 xHundred(55); // 5500
```

http://codepen.io/qualitydixon/pen/XXBwqY

This example has the same functionality as our original but with added flexibility. The outer function is refactored such that it can be reused with a different value for mult. Once again when the nested function is created, it is linked to the data in the current scope, in this case the function parameter mult.

ures are bound to data by reference

Keep in mind closures save the data in the surrounding scope by reference, not by value. Consider an example where two closures are created that share the same environment.

```
function makeOpers() {

 var x = 10;

 return {

    product: function(y) {

       console.log(x*y);

     x++

 },

    sum: function(y) {

       console.log(x+y);

 }

 }

};

var obj = makeOpers();

obj.sum(2); // 12

obj.product(2); // 20

obj.sum(2); // 13
```

http://codepen.io/qualitydixon/pen/JGBQMd

The outer function returns an object containing two functions. The functions print the product or the sum of two values. Both obj.product & obj.sum are closures and contain *references* to the data in their parent environment, which in this case is only the variable x. Additionally, obj.product increments x by one. So when obj.sum is called the second time, the x it 'sees' is now 11, not 10.

This is important for understanding the popular 'closure in a for loop' problem. Here's an example:

```
function printMessage(msg) {

 document.getElementById("text").innerHTML = msg;

}
```

```
function addClicks() {

 for (var i = 1; i <= 3; i++) {

    var btnName = "btn" + i;

    document.getElementById(btnName).onclick = function() {

     printMessage(btnName);

    }

 }

}

addClicks();
```

http://codepen.io/qualitydixon/pen/ZQMbVd

The function addClicks uses a for loop to assign click events to three buttons. Each button is supposed to display a unique text string in an h1 element. Everything may appear to be in working order but look at the result. Each button displays the same

message. Why? Because they all point to the same variable i and by the time the click event is executed, the for loop has completed and i is equal to 3.

Make it private

res are helpful for emulating object oriented programming behavior. ecifically, closures can be used to achieve encapsulation in javascript. To do this, it is necessary to create private members and methods. We've actually already seen how this is done. Look again at the makeOpers example from earlier:

```
function makeOpers() {

 var x = 10;

 return {

    product: function(y) {

        console.log(x*y);

 },

    sum: function(y) {

        console.log(x+y);

 }

 }

};

var obj = makeOpers();

obj.sum(2); // 12

obj.product(2); // 20



console.log(typeof x); // undefined

```

```
33 console.log(typeof obj.x); // undefined
```

http://codepen.io/qualitydixon/pen/bExpNN

x is only accessible through the methods product and sum; it is hidden from the rest program. This gives the developer greater control over how data is accessed manipulated.

Further Reading

It is difficult to overstate the importance of closures in javascript. They are used extensively in many popular web technologies, including nodejs and jquery. For more examples involving closures, check out these posts by mozilla and w3schools.

Mike Dixon

Independent Author and Coder.

February 4, 2016 by Mike Dixon in JavaScript

core language

Comments

Comment

Name

ail

Add Comment