Closing over the loop variable is just as harmful in JavaScript as it is in C#, and more cumbersome to fix

devblogs.microsoft.com/oldnewthing/20140605-00

June 5, 2014

Raymond Chen

Prerequisite reading: Closing over the loop variable considered harmful.

JavaScript has the same problem. Consider:

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  document.getElementById("myButton" + i)
   .addEventListener("click",
         function() { alert(i); });
 }
}
```

The most common case where you encounter this is when you are hooking up event handlers in a loop, so that's the case I used as an example.

No matter which button you click, they all alert `4`, rather than the respective button number.

The reason for this is given in the prerequisite reading: You closed over the loop variable, so by the time the function actually executed, the variable `i` had the value `4`, since that's the leftover value after the loop is complete.

The cumbersome part is fixing the problem. In C#, you can just copy the value to a scoped local and capture the local, but that doesn't work in JavaScript:

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  var j = i;
  document.getElementById("myButton" + i)
   .addEventListener("click",
         function() { alert(j); });
 }
}
```

Now the buttons all alert `3` instead of `4`. The reason is that JavaScript variables have *function* scope, not block scope. Even though you declared `var j` inside a block, the variable's scope is still the entire function. In other words, it's as if you had written

```
function hookupevents() {
 var j;
 for (var i = 0; i < 4; i++) {
  j = i;
  document.getElementById("myButton" + i)
   .addEventListener("click",
         function() { alert(j); });
 }
}
```

Here's a function which emphasizes this "variable declaration hoisting" behavior:

```
function strange() {
 k = 42;
 for (i = 0; i < 4; i++) {
  var k;
  alert(k);
 }
}
```

The function alerts `42` four times because the variable `k` refers to the same variable `k` throughout the entire function, *even before it has been declared*.

That's right. JavaScript lets you use a variable before declaring it.

The scope of JavaScript variables is the function, so if you want to create a variable in a new scope, you have to put it in a new function, since functions define scope.

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  var handlerCreator = function(index) {
   var localIndex = index;
   return function() { alert(localIndex); };
  };
  var handler = handlerCreator(i);
  document.getElementById("myButton" + i)
   .addEventListener("click", handler);
 }
}
```

Okay, now things get weird. We need to put the variable into its own function, so we do that by declaring a helper function `handlerCreator` which creates event handlers. Since we now have a function, we can create a new local variable which is distinct from the variables in the parent function. We'll call that local variable `localIndex`. The handler creator function

saves its parameter in the `localIndex` and then creates and returns the actual handler function, which uses `localIndex` rather than `i` so that it uses the captured value rather than the original variable.

Now that each handler gets a separate copy of `localIndex`, you can see that each one alerts the expected value.

Now, I wrote out the above code the long way for expository purposes. In real life, it's shrunk down quite a bit.

For example, the `index` parameter itself can be used instead of the `localIndex` variable, since parameters can be viewed as merely conveniently-initialized local variables.

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  var handlerCreator = function(index) {
   return function() { alert(index); };
  };
  var handler = handlerCreator(i);
  document.getElementById("myButton" + i)
   .addEventListener("click", handler);
 }
}
```

And then the `handlerCreator` variable can be inlined:

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  var handler = (function(index) {
   return function() { alert(index); })(i);
  document.getElementById("myButton" + i)
   .addEventListener("click", handler);
 }
}
```

And then the `handler` itself can be inlined:

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  document.getElementById("myButton" + i)
   .addEventListener("click",
       (function(index) {
         return function() { alert(index); })(i));
 }
}
```

The pattern `(function (x) { ... })(y)` is misleadingly called the *self-invoking function*. It's misleading because the function doesn't invoke itself; the outer code is invoking the function. A better name for it would be the *immediately-invoked function* since the

function is invoked immediately upon definition.

The next step is to change then the name of the helper `index` variable to simply `i` so that the connection between the outer variable and the inner variable can be made more obvious (and more confusing to the uninitiated):

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  document.getElementById("myButton" + i)
   .addEventListener("click",
       (function(i) {
         return function() { alert(i); })(i));
 }
}
```

The pattern `(function (x) { ... })(x)` is an idiom that means "For the enclosed block of code, capture `x` *by value*." And since functions can have more than one parameter, you can extend the pattern to `(function (x, y, z) { ... })(x, y, z)` to capture multiple variables by value.

It is common to move the entire loop body into the pattern, since you usually refer to the loop variable multiple times, so you may as well capture it just once and reuse the captured value.

```
function hookupevents() {
 for (var i = 0; i < 4; i++) {
  (function(i) {
   document.getElementById("myButton" + i)
    .addEventListener("click", function() { alert(i); });
  })(i);
 }
}
```

Maybe it's a good thing that the fix is more cumbersome in JavaScript. The fix for C# is easier to type, but it is also rather subtle. The JavaScript version is quite explicit.

Exercise: The pattern doesn't work!

```
var o = { a: 1, b: 2 };
document.getElementById("myButton")
 .addEventListener("click",
   (function(o) { alert(o.a); })(o));
o.a = 42;
```

This code alerts `42` instead of `1`, even though I captured `o` by value. Explain.

Bonus reading: C# and ECMAScript approach solving this problem in two different ways. In C# 5, the loop variable of a `foreach` loop is now considered scoped to the loop. ECMAScript code name Harmony proposes a new `let` keyword.

Raymond Chen

Follow