Ev
Th
Br
The Meat
Calculators
Getting Closure
I'm Still Using That!
Never Forget
The Real Life

Never Forget

When I just said *"We're going to change that in the next part"*, did you also take it to mean *"We're going to change the way closures work, so our scripts will fail horribly"*, and not *"We're going to change our perception of closures from something that just works to something we can actually use"*, like it was intended?

I did. It was weird.

Anyway, let's get to it. Have a look at the following piece of code, and think about what it will do, before you actually run it:

RUN ME

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    setTimeout(function () {
      console.log(i);
    }, i * 1000);
  }
};
countSeconds(3);
```

Surprised? There's an inner function and an outer function, and the inner function manages to outlive its outer function by hanging out inside a **`setTimeout`** — and so a closure is formed, just like I described in the previous part. That means that, come time to actually run, the inner function still has access to the outer variable **`i`**, even though the outer function has stopped running and caring about that variable a while ago. No news there. Something's wrong though — whoever wrote that piece of code obviously meant for it to count from 1 to 3, with one second pause between each two counts — all the code spews into the console, though, is **`4`**!

Three-Room Flat

Let's see what's happening: The **`for`**-loop runs three times, and increases **`i`** after every run, so on the fourth try, **`i`** will be 4, and that's the cue for the loop to stop. The closure works alright, so after a second, when the inner function starts doing its thing, it is still able to access **`i`** — which is still 4.

Remember those disgusting locker babies from the previous part? On every run through the loop, one of them is splurted into existence, and they have a whole locker-sharing community thing going on — each of them is forced to use the same stuff, which doesn't make that lipstick thing any more sanitary. What we really want is for each of them to have their own locker, where they can put the loop-index that actually applies to them, so when it's time to run, they still know what it was.

This is the part where we start creating closures intentionally. Have a look at this piece of code:

RUN ME

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    (function () {
      var currentI = i;
      setTimeout(function () {
        console.log(currentI);
      }, currentI * 1000);
    }());
  }
};
countSeconds(3);
```

So, we have wrapped that **`setTimeout`** in an immediately executed function (see those extra parentheses at the end? That means it will run right after it has been declared), and the very first thing it does is put the current value of **`i`** into the local variable **`currentI`**. Now it has its own copy of that value, which will stay just the way it is, no matter what happens to **`i`** in the outer function.

What's important to notice here is that before, a single execution of the outer function produced all three **`setTimeout`**s, so they had the same parent, and thus shared the same locker. Now, though, the parent isn't the outer function any more, it's that immediately executed function (let's call it IEF) we squeezed in there, and since it sits inside the loop, there's one for each of the **`setTimeout`**s. So in the end, each of those guys sits in their own locker, sporting their own copy of **`i`**. And all of those lockers are stacked neatly inside the main locker, where that original **`i`** is still lying around somewhere — but nobody cares, because it is 4, and useless.

Or, to call things by their actual names for a change: Each call to **`setTimeout`** 1.) creates a closure around its own IEF, which has remembered the current value of **`i`** — the value it actually needs later, and 2.) creates a bigger closure around the outer function, so it still has access to those outer variables, too, no matter whether they are needed later or not.

Here's a variation of the code, which uses both the smaller closures around the IEFs, and the bigger ones around the outer function:

RUN ME

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    (function () {
      var currentI = i;
      setTimeout(function () {
        console.log('This is run ' + currentI + ' of ' + howMany + '.');
        console.log('The original i is ' + i + ', which is useless.');
      }, currentI * 1000);
    }());
  }
};
countSeconds(3);
```

Out in he wild, you will probably see this put slightly differently, though, so here are a few variations:

Originally, I've put the local variable (which is supposed to remember the current state) into the IEF using a run-of-the-mill variable declaration. We could easily save some space, though, by putting it in there as a function parameter:

RUN ME

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    // The IEF remembers the current value of i using a function parameter
    (function (currentI) {
      setTimeout(function () {
        console.log(currentI);
      }, currentI * 1000);
    }(i));
  }
};
countSeconds(3);
```

Another thing is that the local variable's name is different from the outer variable's name — I wrote it that way, so it's absolutely clear what's going on — but actually there's no harm in just calling it the same. That means that the outer variable of the same name won't be accessible any more, of course, but we don't need that useless thing anyway. So, this is how that piece of code is usually done:

RUN ME

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    // Here, the outer and the inner variable have the same name.
    // The outer one won't be accessible any more, but that's okay.
    (function (i) {
      setTimeout(function () {
        console.log(i);
      }, i * 1000);
    }(i));
  }
};
countSeconds(3);
```

If you read other people's code, you will also see this variation come up quite often:

RUN ME

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    // The IEF is put right into the setTimeout
    // and returns the actual timeout handler
    setTimeout((function (i) {
      return function () {
        console.log(i);
      };
    }(i)), i * 1000);
  }
};
countSeconds(3);
```

Here, the IEF is put right into **`setTimeout`**. I hope you're still remembering that IEF means "Immediately Executed Function", so just because it's sitting inside **`setTimeout`**, that doesn't mean that it's waiting for anything — it runs *immediately*, and hands the inner function — the one that's actually supposed to wait — over to **`setTimeout`** in the return statement.

This does exactly the same as the previous version, but personally, I prefer the previous one, because I find it much more readable, so I suggest you use that one instead.

In the right circumstances, returning something from an IEF can be very useful, though, and we will see an example of that concept in a minute.

Oops!

Suppose you want to write some sort of animated site navigation, not unlike the one on realLifeJS, where you can use the cursor keys to go to the next or previous page (did you know you could do that here? Go ahead and give it a try!). You will need two functions that do the animating, which you can then call as soon as the right key is pressed:

```
var currentIndex = 0;
var maxIndex = 2;

function next() {
  // If you're already on the last page,
  // just return without doing anything else
  if (currentIndex == maxIndex) return;
  currentIndex++;
  // Animate to page number [currentIndex] here
}
function previous() {
  // If you're already on the first page,
  // just return without doing anything else
  if (currentIndex == 0) return;
  currentIndex--;
  // Animate to page number [currentIndex] here
}
```

That’s very messy though, because it uses two global functions, and whenever you are defining something globally, you are introducing the risk of name conflicts when your code grows — and generally it’s a given that you will totally lose track of what’s what in no time. So, it’s only reasonable to pack those two functions into their own object:

```
var currentIndex = 0;
var maxIndex = 2;

var mainNavigation = {
  next: function () {
    if (currentIndex == maxIndex) return;
    currentIndex++;
    // Animate to page number [currentIndex] here
  },
  previous: function () {
    if (currentIndex == 0) return;
    currentIndex--;
    // Animate to page number [currentIndex] here
  }
};
```

Better. You can call the **`next`** function like **`mainNavigation.next()`** now. Not really ideal, though — there are still those global variables keeping track of the current page index and the total number of pages, and that’s still very messy. Suppose you add some sort of image slideshow on one of your pages; you could put the functions the slideshow needs for going to the next/previous image into their own object, like you just did with the main navigation, but you would have to add new globals for keeping track of the current image and the total number of images, and those globals would probably be called very similar to the ones you already got there, so the whole thing would be very confusing.

So, what to do? You could put everything that has to do with the main navigation into an IEF, so all the stuff that’s currently global would be neatly packed into a local scope:

```
// Everything concerning the main navigation in here:
(function () {
  var currentIndex = 0;
  var maxIndex = 2;

  var mainNavigation = {
    next: function () {
      if (currentIndex == maxIndex) return;
      currentIndex++;
      // Animate to page number [currentIndex] here
    },
    previous: function () {
      if (currentIndex == 0) return;
      currentIndex--;
      // Animate to page number [currentIndex] here
    }
  };
}());
```

That’s no good, though, is it? It’s all wrapped up nicely, so there’s no more potential for conflict or confusion, but it’s wrapped up so tight that you can’t even get to it any more! All of those variables and functions are local to the IEF, and there’s no outside reference to any of them, so they can’t be used at all.

Outside reference, you said? — You know where this is going: 1.) Create an outside reference to those inner functions, 2.) ?, 3.) Profit!

And having read the previous part of this article, you also know that 2.) isn't actually a secret: it's the creation of the closure that happens automatically, as soon as we have done 1.). Here we go:

```
// Everything concerning the main navigation in here:
var mainNavigation = (function () {
  var currentIndex = 0;
  var maxIndex = 2;

  return {
    next: function () {
      if (currentIndex == maxIndex) return;
      currentIndex++;
      // Animate to page number [currentIndex] here
    },
    previous: function () {
      if (currentIndex == 0) return;
      currentIndex--;
      // Animate to page number [currentIndex] here
    }
  };
}());
```

Now that looks nice! The IEF returns the object that's containing the functions we are planning on using in other parts of the code, so that object is what's actually going to end up in the variable **`mainNavigation`**. After that piece of code has run, we can easily call **`mainNavigation.next()`** or **`mainNavigation.previous()`** from anywhere in our code, and those functions will work just fine, because through the power of closures, they still have access to the local variables **`currentIndex`** and **`maxIndex`** – but we don't have to care about them any more. In fact, we *can't* even care about them any more, because they have become completely unavailable to any other part of the code. The only place they still exist in is *inside the closure* we created, and that's exactly the way it should be.

If you have heard anything about OOP (object-oriented programming) yet, you will immediately recognize that we have created an object containing two public methods (the functions in the returned object), which can be called from anywhere, and two private properties (the local variables), which those methods can use as they like, but which can't be accessed from anywhere else. Since just that one object is created, this is actually a singleton pattern, but in the Javascript community, it's much more often referred to as the module pattern.

Of course, we can also have private methods this way: The actual animating part will probably be very similar for **`next`** and **`previous`**, so it makes sense to put it into its own function. This function doesn't have to be accessible from the outside, though, because cursor key navigation won't ever skip a page anyway, so next/previous is all we need publicly. In order to accomplish this, we just have to put a local function in there:

```
// Everything concerning the main navigation in here:
var mainNavigation = (function () {
  var currentIndex = 0;
  var maxIndex = 2;

  var moveTo = function (index) {
    if (index < 0 || index > maxIndex) return;
    currentIndex = index;
    // Animate to page number [currentIndex] here
  }

  return {
    next: function () {
      moveTo(currentIndex + 1);
    },
    previous: function () {
      moveTo(currentIndex - 1);
    }
  };
}());
```

If we were to add some sort of mouse navigation to the cursor key navigation (in real life, this would probably happen the other way round), it's suddenly very possible to go to any arbitrary page at any time, and not just to the next and the previous one. So, the **`moveTo`** method should now actually be made public — and making public methods out of private ones is really easy: We just have to make sure that the returned object (which is the public face of this whole closure construct we have going on here) knows about it:

```
// Everything concerning the main navigation in here:
var mainNavigation = (function () {
  var currentIndex = 0;
  var maxIndex = 2;

  var moveTo = function (index) {
    if (index < 0 || index > maxIndex) return;
    currentIndex = index;
    // Animate to page number [currentIndex] here
  }

  return {
    next: function () {
      moveTo(currentIndex + 1);
    },
    previous: function () {
      moveTo(currentIndex - 1);
    },
    moveTo: moveTo
  };
}());
```

All in all, this is a very neat and sustainable approach, and if you use it to encapsulate different parts of your script that don't have much to do with each other (but in itself build functional units), you stand a fighting chance to keep your house clean and stay on top of things when you start adding more and more functionality to your website.

Round And Round

You've seen two examples now of how intentionally indroducing a bunch of closures to your code can be very helpful. Let's see what they have in common, so you'll be able to take a basic concept away from this, and recognize the situations that beg for the use of closures:

First, the **`setTimeout`** thing: Here, the closure was used to *remember the current loop index*, so it could be used later on. There wasn't really any other place we could have stored away that value for later use, so it was pretty damn lucky we had that closure idea.

Second, the module pattern: Here, the closure was used to *remember all sorts of stuff* that the public methods needed in order to do their thing later on. We could have stored away that stuff globally, but that would have been pretty messy, since no other part of the code was actually going to use any of it.

So, it's pretty obvious what's going on: you can use closures to remember stuff for later use, without having to put it somewhere where it doesn't actually belong.

Having understood that concept, it shouldn't be too hard to dream up situations in which you can use it. Just to give you an idea: Suppose you're doing a lot of rounding in some sort of calculating application, and for debugging purposes, you want to get some logging output to the console whenever a number is rounded. You *could* track down every **`Math.round`** in your code and add a **`console.log`** in all of those places, but that's just silly —

you can simply overwrite **`Math.round`** itself, so it does the logging for you! There's a problem, though: your own custom version of **`Math.round`** still needs to call the original **`Math.round`**, so the actual calculations are still being done — but that original function won't be available under that name any more, after you've overwritten it. That means you have to *remember* the original function, so it can be used later on by the overwritten one.

That's exactly the situation we were talking about, and this can be done easily by creating a closure over an IEF that takes the original **`Math.round`** in as a parameter:

RUN ME

```
(function (originalFunction) {
  Math.round = function (number) {
    var result = originalFunction.call(this, number);
    console.log(number + ' has been rounded to ' + result);
    return result;
  };
}(Math.round));

alert(Math.round(Math.PI));
```

We could have stored away the original function globally, but why would we? This approach gives you a neat self-contained snippet that doesn't leave any turds in the global namespace and can be dropped in and out of your scripts without any risk of messing with any other parts of your code.

It takes a bit of practice to see where a closure can help you, but keeping that basic concept of remembering for later use in mind, you'll figure it out in no time.

Next up, a few full-page examples of closures in action.

Next Part