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

I'm Still Using That!

Okay, here we go. Consider the following boring piece of code:

```
function myFunction() {
  var myVar = 1;
  var alertMyVar = function () {
    alert('My variable has the value ' + myVar);
  };
  alertMyVar();
}
myFunction();
```

Question: After running those eight lines of code, what value will the variable **myVar** you see in there have?

The simple answer: What do you care? It's a local variable defined inside **myFunction**, and as you might know, variables defined inside a function can't be accessed from the outside. So, whatever its value, you can't get to it anyway.

A slightly more involved answer: Since that variable is trapped (scoped) inside a function, and there's no way to get to it after that function has finished, asking for its value is a bit like asking, "If Helen Keller falls in the woods, does she make a sound?" Fact of the matter is, though, that using a physical definition of "sound", she probably does. So, what you can do is try to answer the question on a lower level: You can't use that variable any more, but its value must have been stored in *some* place during the execution of **myFunction**, and it might even still be there. Okay, so is it still there, somewhere, in the depths of your Computer's memory banks? The truth is, it might or might not be. This depends on whether the garbage truck has already been there to haul away all the stuff you can't use any more — and when exactly those garbage trucks are being sent out is at the discretion of the Javascript implementation (meaning, it's up to your browser).

So, this example hasn't been all that informative, and those garbage trucks don't really have to concern you. But there's at least *something* you can take away from it: If nobody can remember the combination to a locker, it doesn't really matter what's inside.

Vorfreude: A Play in One Act

Now, consider this:

```
function myFunction() {
  var myVar = 1;
  var alertMyVar = function () {
    alert('My variable has the value ' + myVar);
  };
  setTimeout(alertMyVar, 1000 * 3600 * 24);
}
myFunction();
```

Question: Same as before — after running those eight lines of code, what's the value of the variable **myVar**?

Answer: It's 1, of course; what else would it be? The more interesting question is, why is this an easy answer now — what's different? After the

code has run, you still can't use the local variable `myVar` *outside* of `myFunction` in order to get to its value — but there's that delayed `alertMyVar` function camping *right in the middle* of `myFunction`, just waiting for his time to shine – and he's obviously still using `myVar`. So, *you* can't get to `myVar`, just like before, but *someone* can, and that's enough to hold off the garbage trucks: `myFunction` has finished doing its thing and is ready to throw out the trash, but `alertMyVar` steps up briskly and says,

alertMyVar: "Are you *kidding* me with those trash bags? I'm already pissed because I'm supposed to sit around here for a *day* and do absolutely *nothing*, just so I can give those jokers the value of that variable this time tomorrow — and you got nothing better to do than to *throw it out?!*"

myFunction: "*Fuck you*, I'm done for today. You love that trash so much, you take it."

alertMyVar: "Alright, just give it already. Now get outta here!"

`alertMyVar` reaches into his fanny pack, pulls out a huge archaic looking bell alarm clock, puts it down on the concrete floor, lies down beside it, and starts twiddling his thumbs, eyes wide open. It's a long wait, but the thought of his moment in the spotlight keeps `alertMyVar` going. He's imagining it over and over: jumping up with a huge TA-DAAAA-like gesture, screaming at the top of his lungs, "My variable has the value …" — well, what value does it have, actually? For a moment he's considering opening the trash bag he got from `myFunction` (and which he is using as a pillow right now) and having a peek, but he quickly reconsiders; the value might still change — variables are fickle fellows that way — and he needs to avoid the risk of mixing something up in his head once the big moment has finally arrived.

He doesn't realize that he has nearly fallen asleep, until he hears a sharp noise. Is that it? He jerks around to look at his alarm clock, and freezes up in shock. It's just sitting there, ticking away, with more than 23 hours to go — and he knows full well what that means: The noise he was hearing was the door bell ringing. It's the garbage men, and they are coming for him. He should have realized that no User has the patience to wait around for a day. His hands tremble a bit as he lies back down. Everybody knows that getting your tab shut down on you is the number one cause of premature demise around here, but nobody ever seriously considers this happening to *them*.

There's a violent noise pushing through his ears and into his brain, and even before he is able to see the large chunks of wood flying by above his head at a dangerously low height, he spreads his arms, exhales all the burdens of existence, and closes his eyes. Just a moment later, right when the pieces of what once was the door come crashing into the wall at the other side of the room and, for a moment, start forming a distorted door-mosaic there before deciding to abandon their Platonic prototype altogether and instead transform into a million splinters glistening orange in the sunlight, which is now flowing through the large rectangular hole in the wall, competing with the wooden shrapnel to fill up the stale air — just at that moment, and, remarkably, over all the noise that's currently penetrating his (at least comfortably cushioned) head, he hears the rustling sound of a huge trash bag being opened above his head. Completely letting go now, he mutters a last feeble "fuck", unaware that no man in this most absolute of situations ever didn't.

And that, my friends, is what a closure does: It makes sure that whenever function B sits inside function A and needs to use one of A's variables, it just

gets to use it, no questions asked, even if A has already done everything it's supposed to do and gone home for the day.

Death and Lipstick

In case that story made you sad (it didn't actually turn out to be a play, now, did it? I hope you at least imagined the non-dialogue parts being read out by Morgan Freeman), here's a different one … remember the thing about the locker I told you to take away from the first example? Suppose every person has their own personal locker, which holds all of their stuff. If someone dies, the combination usually dies with them, and that's that. No one can open the locker any more, so it's just thrown out to the trash.

Some people have children, though, which, for convenience's sake, they keep in their locker too. This also works out great for the kid: whenever little Klaus wants to stick Mommy's lipstick right up his nose, he can just go ahead and do it, no questions asked. One day, though — *BAM* — she dies ("going home for the day" was a euphemism for dying all along, you know) and her locker gets hauled away with everything in it, including that poor little creep, never to be seen again:

```
// This is Mommy
function MargareteKüblinger() {
  // Here's all the stuff she keeps in her locker:
  var lipstick = "[===]=>"; // Delicious lipstick
  var KlausKüblinger = function () { // That's me, Klaus
    alert('I enjoy a good piece of ' + lipstick + ' up my nose.');
  };
  // This is where mommy first told me to nasally abuse her lipstick.
  // She's the real creep here, you know.
  KlausKüblinger();
}

// This is where it all started
MargareteKüblinger();

// At this point, Mommy died. The next day they came for the locker.
// I was hoping I could see her again on the other side.
// No such luck. I live in Limbo now. It sucks.
```

Now, not all parents are shit, and there are at least some that let their kids out of the locker at some point and send them off to school or something. Since the kids might need a thing or two out of the locker they've been living in up until now, they just give them the combination. When the parent dies, the kid still has that combination and can continue to use all the stuff. And since somebody is still using the locker, nobody will dare throwing it out to the trash:

```
// This is my school
var school = {};

// This is Mommy
function MargareteKüblinger() {
  // Here's all the stuff she keeps in her locker:
  var lipstick = "[===]=>"; // Delicious lipstick
  var KlausKüblinger = function () { // That's me, Klaus
    alert('I enjoy a good piece of ' + lipstick + ' up my nose.');
  };
  // This is where mommy first told me to nasally abuse her lipstick.
  // She's the real creep here, you know.
  KlausKüblinger();
  // This is where Mommy decided to send me off to school:
  school.studentNumber655321 = KlausKüblinger;
}

// This is where it all started
MargareteKüblinger();

// At this point, Mommy died. I was at school when it happened. What a bummer!
```

```
22
23 // This is where I again felt the urge to do that disgusting thing:
24 school.studentNumber655321();
25 // Since Mommy has given me the combination to her locker
26 // before sending me off, I just went ahead and indulged.
27 // What a feeling! I can still smell it in my brain.
```

Every Metaphor Has A Breaking Point

Okay, so here's the deal (the code snippets illustrating the story have probably already given away most of this):

1. Those people are functions.
2. B being a child of A means that the function B is nested inside of the function A.
3. The stuff in the locker are the local variables of function A.
4. The "locker" being "thrown out" is Javascript's garbage collection clearing the memory that has been used by function A's local variables (function B being one of them).
5. "Dying" for a function means "having finished running from start to end".
6. "Letting the kid out of the locker" means creating a reference to the inner function (B) outside of the outer function (A).
7. "Giving the kid the combination to the locker" means — *drumroll* — creating a closure.

Most of this should be rather clear — point 6 might need some explanation, though: What exactly does "creating a reference to the inner function outside of the outer function" encompass? You've already seen two examples:

First, the outer function can call the inner function after some delay by setting up a timing event (using **`setTimeout`** or **`setInterval`**). Actually, it doesn't call the inner function at all, it just gives it to the timer and tells it when to call it. For the timer to be able to do so, it has to keep its own reference to the function it's supposed to call. You've already seen that:

```
1  // Creating a closure using timing events
2  function myFunction() {
3    var myVar = 1;
4    var alertMyVar = function () {
5      alert('My variable has the value ' + myVar);
6    };
7    // setTimeout creates an outside reference to alertMyVar,
8    // so it will be able to call it after a day has passed
9    setTimeout(alertMyVar, 1000 * 3600 * 24);
10 }
11 myFunction();
```

Second, the outer function can create such a reference explicitly, like **`school.studentNumber655321 = KlausKüblinger`**. In this case, **`school.studentNumber655321`** is that outside reference we're talking about here (notice how the object **`school`** is defined *outside* of the outer function **`Mommy`**). In that snippet above, **`school`** was even *global*, but that's not necessary. Here's another example:

```
1  // Creating a closure by explicitly creating an outside reference
2  // from within the containing function
3  var alertMyVarGlobal;
4  function myFunction() {
5    var myVar = 1;
6    var alertMyVar = function () {
7      alert('My variable has the value ' + myVar);
8    };
9    // The inner function alertMyVar is stored in the global variable alertMyVarGlobal
10   // which means that alertMyVarGlobal is now an outside reference
```

```
  alertMyVarGlobal = alertMyVar;
}
myFunction();
```

There are two other ways, which are very similar to the two above, though:

Third, the outer function can set up the inner function to handle a DOM event (which is, in a way, quite similar to the timing events mentioned earlier). There are quite a few of those, among them the popular **`onload`**, **`onclick`**, and **`onmouseover`**. Just like with the timing events, the browser's event dispatcher keeps an outside reference to the function you registered as an event handler, so it can call it whenever the event is triggered.

```
// Creating a closure by making the inner function an event handler
function myFunction() {
  var myVar = 1;
  var alertMyVar = function () {
    alert('My variable has the value ' + myVar);
  };
  var body = document.getElementsByTagName('body')[0];
  // alertMyVar is set up as an event handler
  // In this case (DOM level 0 event model), this explicitly
  // creates an outside reference
  body.onclick = alertMyVar;
  // In this case, on the other hand (DOM level 2 event model),
  // the outside reference is created by addEventListener/attachEvent
  body.addEventListener ?
    body.addEventListener('click', alertMyVar, false) :
    body.attachEvent('onclick', alertMyVar);
}
myFunction();
```

Fourth, the outer function can return the inner function, or pass it off to some other function as a parameter — and then whoever receives that return value/function parameter has to do one of those other three things already listed to create the outside reference. This is quite similar to the second one, in that the inner function is being passed off explicitly to some other piece of code, the difference being that the outer function doesn't create the reference itself any more, but lets that other piece of code decide how (or whether) to do that:

```
// Creating a closure by returning the inner function,
// and using that return value to create an outside reference
function myFunction() {
  var myVar = 1;
  var alertMyVar = function () {
    alert('My variable has the value ' + myVar);
  };
  return alertMyVar;
}
var outsideReference = myFunction();
// outsideReference is now a reference to alertMyVar outside of myFunction
```

And that's all there is. Those are the only ways to create closures.

Finally, revisiting those ghastly locker babies one last time, what's interesting to note is that *every time* one of them is let out, it gets the combination to the locker. There's no way around it. In other words, whenever a reference to an inner function is created outside of its containing function, a closure is created. *Always*. So, to put it differently once more, whenever some function outlives the function it's sitting in (because it got referenced from the outside), it invariably gets to keep access to the variables it has been accessing all along. Nothing changes at all.

And that's exactly the reason why beginners often don't "get" closures: On first glance, they don't seem like anything you could actually *use* — just something that works in the background to keep your scripts from failing

horribly.

We're going to change that in the next part.

Next Part