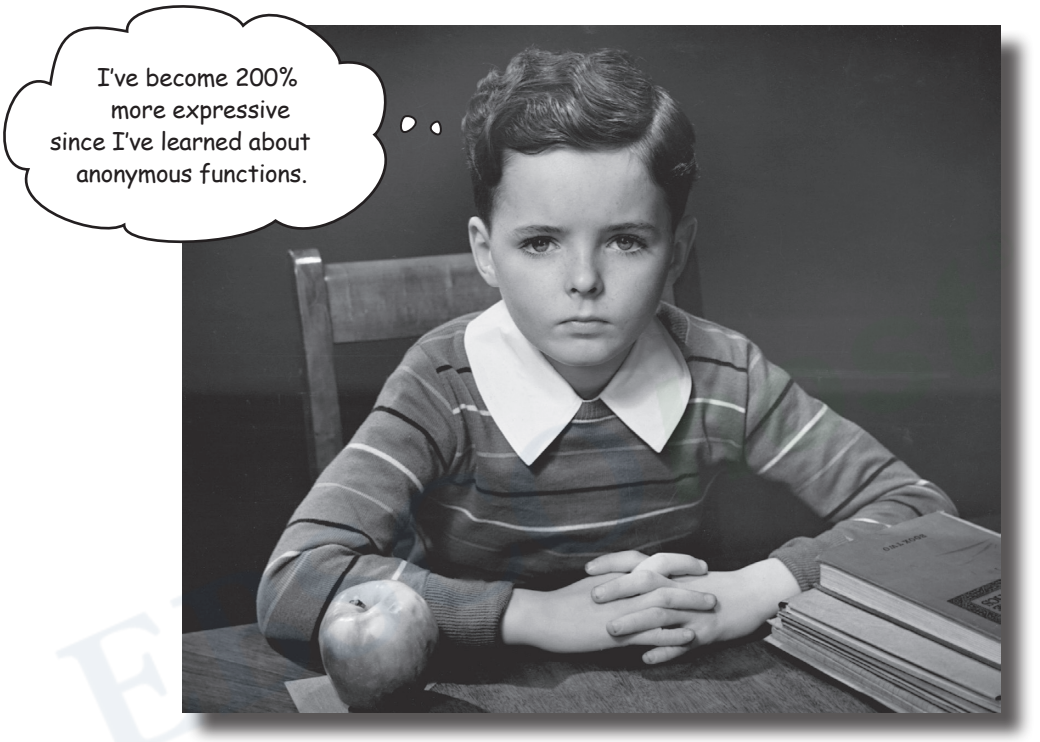
11 anonymous functions, scope and closures

Serious functions



You've put functions through their paces, but there's more to learn. In this chapter we take it further; we get hard-core. We're going to show you how to **really handle** functions. This won't be a super long chapter, but it will be intense, and at the end you're going to be more expressive with your JavaScript than you thought possible. You're also going to be ready to take on a coworker's code, or jump into an open source JavasScript library, because we're going to cover some common coding idioms and conventions around functions. And if you've never heard of an **anonymous function** or a **closure**, boy are you in the right place.

And if you have heard of a closure, but don't quite know what it is, you're even more in the right place!

Taking a look at the other side of functions...

You've already seen two sides of functions—you've seen the formal, declarative side of function declarations, and you've seen the looser, more expressive side of function expressions. Well, now it's time to introduce you to another interesting side of functions: *the anonymous side.*

By anonymous we're referring to functions *that don't have names.* How can that happen? Well, when you define a function with a function declaration, your function will *definitely have a name*. But when you define a function using a function expression, *you don't have to give that function a name*.

You're probably saying, sure, that's an interesting fact, maybe it's possible, but so what? By using anonymous functions we can often make our code less verbose, more concise, more readable, more efficient, and even more maintainable.

So let's see how to create and use anonymous functions. We'll start with a piece of code we've seen before, and see how an anonymous function might help out:

Here's a load handler, set up like we've always done in the past.

First we define a function. This function has a name, handler.

```
function handler() { alert("Yeah, that page loaded!"); }
window.onload = handler;
```

Then we assign the function to the onload property of the window object, using its name, handler.

And when the page loads, the handler function is invoked.

Sharpen your pencil

Use your knowledge of functions and variables and check off the true statements below.

- ❑ The handler variable holds a function reference.
- ❑ When we assign handler to window.onload, we're assigning it a function reference.
- ❑ The only reason the handler variable exists is to assign it to window.onload.
- ❑ We'll never use handler again as it's code that is meant to run only when the page first loads.
- ❑ Invoking onload handlers twice is not a great idea—doing so could cause issues given these handlers usually do some initialization for the entire page.
- ❑ Function expressions create function references.
- ❑ Did we mention that when we assign handler to window.onload, we're assigning it a function reference?

How to use an anonymous function

So, we're creating a function to handle the load event, but we know it's a "one time" function because the load event only happens once per page load. We can also observe that the `window.onload` property is being assigned a function reference—namely, the function reference in `handler`. But because `handler` is a one time function, that name is a bit of a waste, because all we do is assign the reference in it to the `window.onload` property.

Anonymous functions give us a way to clean up this code. An anonymous function is just a function expression without a name that's used where we'd normally use a function reference. But to make the connection, it helps to see how we use a function expression in code in an anonymous way:

```
function handler() { alert("Yeah, that page loaded!"); }
window.onload = handler;
```

First remove the handler variable so this becomes a function expression.

```
window.onload = function() { alert("Yeah, that page loaded!"); };
```

Then assign the function expression directly to the window.onload property.

```
window.onload = function() { alert("Yeah, that page loaded!"); };
```

Now the handler has been assigned to the window.onload property without the need for an unnecessary name.

Much more concise now. We're assigning the function we need directly to the onload property. We also aren't creating a function name that might mistakenly be used in other code (after all, handler is a pretty common name!).

Look Ma! No names!

BRAIN POWER

Are there places in your previous code that you've seen anonymous functions and hadn't realized it?

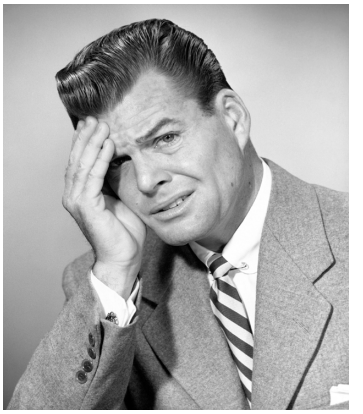
Hint: are they hiding somewhere in your objects?

Sharpen your pencil

There are a few opportunities in the code below to take advantage of anonymous functions. Go ahead and rework the code to use anonymous functions wherever possible. You can scratch out the old code and write in new code where needed. Oh, and one more task: circle any anonymous functions that are already being used in the code.

```
window.onload = init;
var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
              console.log("Baking the cookies.");
              setTimeout(done, time);
          }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
}
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

We need to talk about your verbosity, again



Okay, we hate to bring it up again because you've come a long way with functions—you know how to pass functions around, assign them to variables, pass them to functions, return them from functions—but, well, you're still being a little more verbose than you have to (you could also say you're not being as expressive as you could be). Let's see an example:

Here's a normal-looking function named cookieAlarm that displays an alert about cookies being done.

```
function cookieAlarm() {
    alert("Time to take the cookies out of the oven");
};

setTimeout(cookieAlarm, 600000);
```

Looks like the cookies will be done in 10 minutes, just sayin'.

In case you forgot, these are milliseconds, so 1000 * 60 * 10 = 600,000.

And here we're taking the function and passing it as an argument to setTimeout.

While this code looks fine, we can make it a bit tighter using anonymous functions. How? Well, think about the `cookieAlarm` variable in the call to `setTimeout`. This is a variable that references a function, so when we invoke `setTimeout`, a function reference is passed. Now, using a variable that references a function is one way to get a function reference, but just like with the `window.onload` example a couple of pages back, you can use a function expression too. Let's rewrite the code with a function expression instead:

Now instead of a variable, we're just putting the function, inline, in the call to setTimeout.

Pay careful attention to the syntax here. We write the entire function expression, which ends in a right bracket, and then like any argument, we follow it with a comma before adding the next argument.

```
setTimeout(function() { alert("Time to take the cookies out of the oven");}, 600000);
```

We specify the name of the function we're calling, setTimeout, followed by a parenthesis and then the first argument, a function expression.

Here's the second argument, after the function expression.

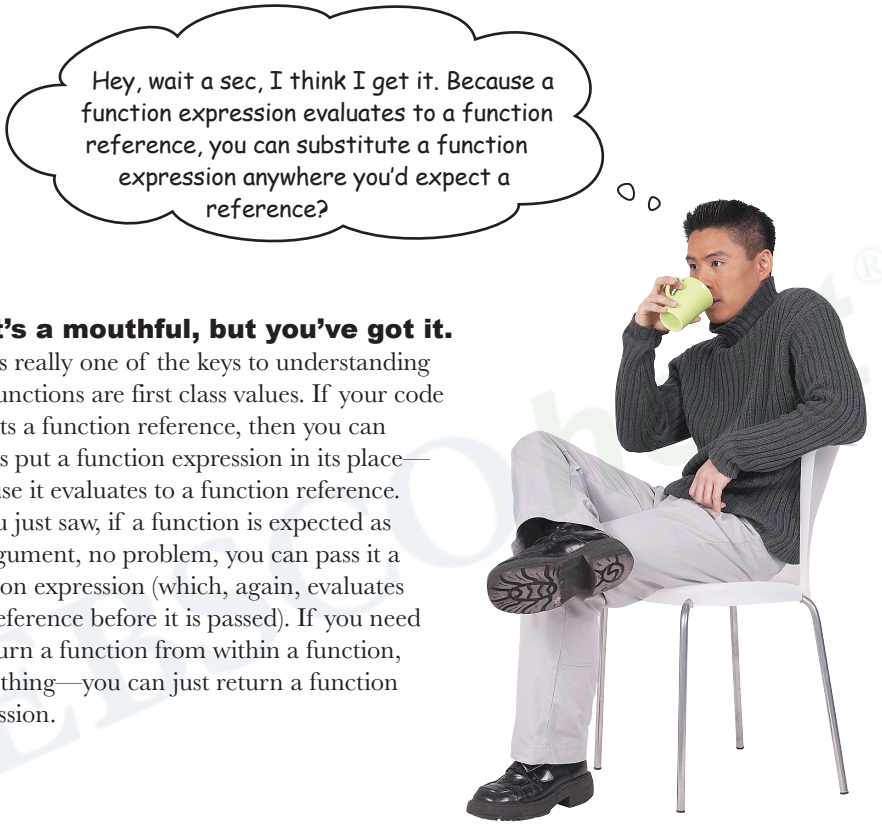
Who are you trying to kid? That's a mess. Who wants to read that one long line? And what if the function is long and complicated?

For a short piece of code, a one liner is just fine. But, beyond that, you're right, it would be rather silly. But as you know, we can use lots of whitespace with JavaScript, so we can insert all the spaces and returns we need to make things more readable. Here's our reformatting of the `setTimeout` code on the previous page.

All we've done is insert some whitespace—that is, some spaces and returns here and there.

```
setTimeout(function() {
             alert("Time to take the cookies out of the oven");
           }, 600000);
```

We're glad you raised this issue because the code is a lot more readable this way.



That's a mouthful, but you've got it. This is really one of the keys to understanding that functions are first class values. If your code expects a function reference, then you can always put a function expression in its place—because it evaluates to a function reference. As you just saw, if a function is expected as an argument, no problem, you can pass it a function expression (which, again, evaluates to a reference before it is passed). If you need to return a function from within a function, same thing—you can just return a function expression.

Let's make sure you have the syntax down for passing anonymous function expressions to other functions. Convert this code from one that uses a variable (in this case vaccine) as an argument to one that uses an anonymous function expression.

```
function vaccine(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}

administer(patient, vaccine, time);
```

Write your version here. And check your answer before moving on!

there are no Dumb Questions

Q: Using these anonymous functions like this seems really esoteric. Do I really need to know this stuff?

A: You do. Anonymous function expressions are used frequently in JavaScript code, so if you want to be able to read other people's code, or understand JavaScript libraries, you're going to need to know how these work, and how to recognize them when they're being used.

Q: Is using an anonymous function expression really better? I think it just complicates the code and makes the code hard to follow and read.

A: Give it some time. Over time, you'll be able to parse code like this more easily when you see it, and there really are lots of cases where this syntax decreases code complexity, makes the code's intention more clear, and cleans up your code. That said, overuse of this technique can definitely lead to code that is quite hard to understand. But stick with it and it'll get easier to read and more useful as you get the hang of it. You're going to encounter lots of code that makes heavy use of anonymous functions, so it's a good idea to incorporate this technique into your code toolbelt.

Q: If first class functions are so useful, how come other languages don't have them?

A: Ah, but they do (and even the ones that don't are considering adding them). For instance, languages like Scheme and Scala have fully first class functions like JavaScript does. Other languages, like PHP, Java (in the newest version), C#, and Objective C have some or most of the first class features that JavaScript does. As more people are recognizing the value of having first class functions in a programming language, more languages are supporting them. Each language does it a little differently, however, so be prepared for a variety of approaches as you explore this topic in other languages.

When is a function defined? It depends...

There's one fine point related to functions that we haven't mentioned yet. Remember that the browser takes two passes through your JavaScript code: in the first pass, all your function declarations are parsed and the functions defined; in the second pass, the browser executes your code top down, which is when function expressions are defined. Because of this, functions created by declarations are defined *before* functions that are created using function expressions. And this, in turn, determines where and when you can invoke a function in your code.

To see what that really means, let's take a look at a concrete example. Here's our code from the last chapter, rearranged just a bit. Let's evaluate it:

IMPORTANT: Read this by following the order of the numbers. Start at 1, then go to 2, and so on.

1 We start at the top of the code and find all the function declarations.

4 We start at the top again, this time evaluating the code.

```
var migrating = true;
```

5 Create the variable migrating and set it to true.

Notice that we moved this conditional up from the bottom of the code.

```
if (migrating) {
    quack(4);
    fly(4);
}
```

6 The conditional is true, so evaluate the code block.

7 Get the function reference from quack and invoke it with the argument 4.

8 Get the function reference from fly... oh wait, fly isn't defined!

```
var fly = function(num) {
  for (i = 0; i < num; i++) {
    console.log("Flying!");
  }
};
```

2 We found a function declaration. We create the function and assign it to the variable quack.

```
function quack(num) {
  for (i = 0; i < num; i++) {
    console.log("Quack!");
  }
}
```

3 We reach the bottom. Only one function declaration was found.

What just happened? Why wasn't fly defined?

Okay, we know the `fly` function is undefined when we try to invoke it, but why? After all, `quack` worked just fine. Well, as you've probably guessed by now, unlike `quack`—which is defined on the first pass through the code because it is a function declaration—the `fly` function is defined along with the normal top-to-bottom evaluation of the code. Let's take another look:

When we evaluate this code to try invoking quack, everything works as expected because quack was defined on the first pass through the code.

```
var migrating = true;
if (migrating) {
    quack(4);
    fly(4);
}
var fly = function(num) {
   for (var i = 0; i < num; i++) {
      console.log("Flying!");
   }
};
function quack(num) {
   for (var i = 0; i < num; i++) {
      console.log("Quack!");
   }
}
```

But when we try to execute the call to the fly function, we get an error because we haven't yet defined fly...

...because fly doesn't get defined until this statement is evaluated, which is after the call to fly.

JavaScript console

```
Quack!
Quack!
Quack!
Quack!
TypeError: undefined is not a function
```

What happens when you try to call a function that's undefined.

You might see an error like this instead (depending on the browser you're using): TypeError: Property 'fly' of object [object Object] is not a function.

So what does this all mean? For starters, it means that you can place function declarations anywhere in your code—at the top, at the bottom, in the middle—and invoke them wherever you like. Function declarations create functions that are defined everywhere in your code (this is known as *hoisting*).

Function expressions are obviously different because they aren't defined until they are evaluated. So, even if you assign the function expression to a global variable, like we did with `fly`, you can't use that variable to invoke a function until after it's been defined.

Now in this example, both of our functions have *global scope*—meaning both functions are visible everywhere in your code once they are defined. But we also need to consider nested functions—that is functions defined within other functions—because it affects the scope of those functions. Let's take a look.

How to nest functions

It's perfectly legal to define a function within another function, meaning you can use a function declaration or expression inside another function. How does this work? Here's the short answer: the only difference between a function defined at the top level of your code and one that's defined within another function is just a matter of scope. In other words, placing a function in another function affects where the function is visible within your code.

To understand this, let's expand our example a little by adding some nested function declarations and expressions.

```
var migrating = true;
var fly = function(num) {
  var sound = "Flying";
  function wingFlapper() {
    console.log(sound);
  }
  for (var i = 0; i < num; i++) {
    wingFlapper();
  }
};
function quack(num) {
  var sound = "Quack";
  var quacker = function() {
    console.log(sound);
  };
  for (var i = 0; i < num; i++) {
    quacker();
  }
}
if (migrating) {
  quack(4);
  fly(4);
}
```

Here we're adding a function declaration with the name wingFlapper inside the fly function expression.

And here we're calling it.

Here we're adding a function expression assigned to the quacker variable inside the quack function declaration.

And here we're calling it.

We've moved this code back to the bottom so we no longer get that error when we call fly.

Exercise

In the code above, take a pencil and mark where you think the scope of the fly, quack, wingFlapper and quacker functions are. Also, mark any places you think the functions might be in scope but undefined.

How nesting affects scope

Functions defined at the top level of your code have global scope, whereas functions defined within another function have local scope. Let's make a pass over this code and look at the scope of each function. While we're at it, we'll also look at where each function is defined (or, not `undefined`, if you prefer):

```
var migrating = true;
var fly = function(num) {
   var sound = "Flying";
   function wingFlapper() {
      console.log(sound);
   }
   for (var i = 0; i < num; i++) {
      wingFlapper();
   }
};
function quack(num) {
   var sound = "Quack";
   var quacker = function() {
      console.log(sound);
   };
   for (var i = 0; i < num; i++) {
      quacker();
   }
}
if (migrating) {
    quack(4);
    fly(4);
}
```

Everything defined at the top level of the code has global scope. So fly and quack are both global variables.

But remember fly is defined only after this function expression is evaluated.

wingFlapper is defined by a function declaration in the fly function. So its scope is the entire fly function, and it's defined throughout the entire fly function body.

quacker is defined by a function expression in the function quack. So its scope is the entire quack function but it's defined only after the function expression is evaluated, until the end of the function body.

quacker is only defined here.

Notice that the rules for when you can refer to a function are the same within a function as they are at the top level. That is, within a function, if you define a nested function *with a declaration*, that nested function is defined everywhere within the body of the function. On the other hand, if you create a nested function using a *function expression*, then that nested function is defined only after the function expression is evaluated.

there are no Dumb Questions

Q: When we pass a function expression to another function, that function must get stored in a parameter, and then treated as a local variable in the function we passed it to. Is that right?

A: That's exactly right. Passing a function as an argument to another function copies the function reference we're passing into a parameter variable in the function we've called. And just like any other parameter, a parameter holding a function reference is a local variable.

EXTREME JAVASCRIPT CHALLENGE

We need a first class functions expert and we've heard that's you! Below you'll find two pieces of code, and we need your help figuring out what this code does. We're stumped. To us, these look like nearly identical pieces of code, except that one uses a first class function and the other doesn't. Knowing everything we do about JavaScript scope, we expected Specimen #1 to evaluate to 008 and Specimen #2 to evaluate to 007. But they both result in 008! Can you help us figure out why?

We recommend you form a strong opinion, jot it down on this page, and then turn the page.

Specimen #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}

getSecret();
```

Specimen #2

```
var secret = "007";

function getSecret() {
  var secret = "008";

  function getValue() {
    return secret;
  }
  return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Don't look at the solution at the end of the chapter just yet; we'll revisit this challenge a little bit later.

A little review of lexical scope

Lexical just means you can determine the scope of a variable by reading the structure of the code, as opposed to waiting until the code runs to figure it out.

While we're on the topic of scope, let's quickly review how lexical scope works:

Here we have a global variable called justAVar.

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    return justAVar;
}

var result = whereAreYou();
console.log(result);
```

And this function defines a new lexical scope...

...in which we have a local variable, justAVar, that shadows the global variable of the same name.

When this function is called, it returns justAVar. But which one? We're using lexical scope, so we find the justAVar value by looking in the nearest function scope. And if we can't find it there, we look in the global scope.

So when we call whereAreYou, it returns the value of the local justAVar, not the global one.

JavaScript console

```
Just an every day LOCAL
```

Now let's introduce a nested function:

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    function inner() {
        return justAVar;
    }

    return inner();
}

var result = whereAreYou();
console.log(result);
```

Here's the same function.

And shadow variable.

But now we have a nested function, that refers to justAVar. But which one? Well, again, we always use the variable from the closest enclosing function. So we're using the same variable as the last time.

Notice that we're calling inner here, and returning its result..

So when we call whereAreYou, the inner function is invoked, and returns the value of the local justAVar, not the global one.

JavaScript console

```
Just an every day LOCAL
```

Where things get interesting with lexical scope

Let's make one more tweak. Watch this step carefully; it's a doozy:

```
var justAVar = "Oh, don't you worry about it, I'm GLOBAL";

function whereAreYou() {
    var justAVar = "Just an every day LOCAL";

    function inner() {
        return justAVar;
    }

    return inner;
}

var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

No changes at all here, same variables and functions.

But rather than invoking inner, we return the inner function.

So when we call whereAreYou, we get back a reference to inner function, which we assign to the innerFunction variable. Then we invoke innerFunction, capture its output in result and display the result.

So when inner is invoked here (as innerFunction), which justAVar is used? The local one, or the global one?

What matters is when the function is invoked. We invoke inner after it's returned, when the global version of justAVar is in scope, so we'll get "Oh don't worry about it, I'm GLOBAL".

Not so fast. With lexical scope what matters is the structure in which the function is defined, so the result has to be the value of the local variable, or "Just an everyday LOCAL".

Frank: What do you mean you're right? That's like defying the laws of physics or something. The local variable doesn't even exist anymore… I mean, when a variable goes out of scope it ceases to exist. It's derezzed! Didn't you see TRON!?

Judy: Maybe in your weak little C++ and Java languages, but not in JavaScript.

Jim: Seriously, how is that possible? The `whereAreYou` function has come and gone, and the local version of `justAVar` couldn't possibly exist anymore.

Judy: If you'd listen to what I just told you… In JavaScript that's not how it works.

Frank: Well, throw us a bone Judy. How does it work?

Judy: When we define the `inner` function, the local `justAVar` is in the scope of that function. Now lexical scope says how we define things is what matters, so if we're using lexical scope, then *whenever* `inner` is invoked, it assumes it still has that local variable around if it needs it.

Frank: Yeah, but like I already said, that's like defying the laws of physics. The `whereAreYou` function that defined the local version of the `justAVar` variable is over. It doesn't exist any more.

Judy: True. The `whereAreYou` function is done, but the scope is still around for `inner` to use.

Jim: How is that?

Judy: Well, let's see what REALLY happens when we define and return a function…

EDITOR'S NOTE: Did Joe change his shirt between pages!?

Functions Revisited

We have a bit of a confession to make. Up until now we haven't told you *everything* about a function. Even when you asked "What does a function reference actually point to?" we kinda skirted the issue. "Oh just think of it like a crystallized function that holds the function's code block," we said.

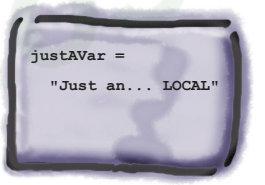
Well now it's time to show you everything.

To do that, let's walk through what really happens at runtime with this code, starting with the `whereAreYou` function:

```
function whereAreYou() {

    var justAVar = "Just an every day LOCAL";

    function inner() {
        return justAVar;
    }

    return inner;
}
```

1 First we encounter a local variable, named justAVar. We assign the string "Just an every day LOCAL" to the variable.

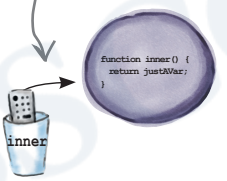
2 We haven't mentioned this before, but all local variables are stored in an environment.



This is the environment. It holds all the variables defined in the local scope.

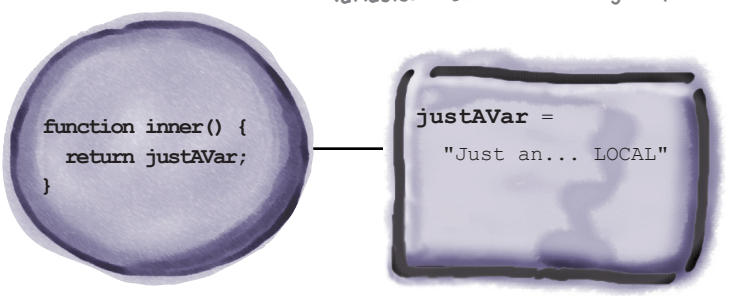
In this example, the environment just has one variable, the justAVar variable.

3 We then create a function, with the name inner.



4 And finally, when we return the function, we don't just return the function; we return the function with the environment attached to it.

Every function has an attached environment, which contains the local variables within its enclosing scope.



So, let's see how this gets used when we call the function inner...

Calling a function (revisited)

Now that we have the `inner` function, and its environment, let's invoke `inner` and see what happens. Here's the code we want to evaluate:

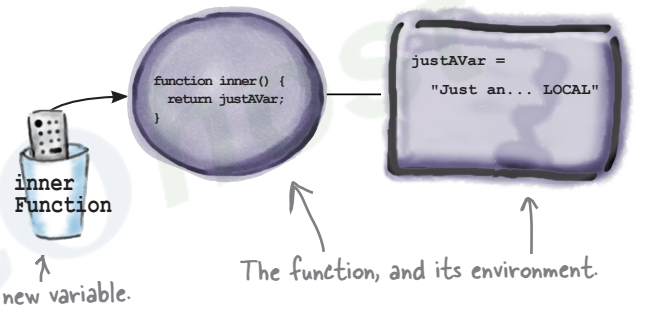
```
var innerFunction = whereAreYou();
var result = innerFunction();
console.log(result);
```

---

1 **First, we call whereAreYou. We already know that returns a function reference. So we create a variable innerFunction and assign it that function. Remember, that function reference is linked to an environment.**

```
var innerFunction = whereAreYou();
```

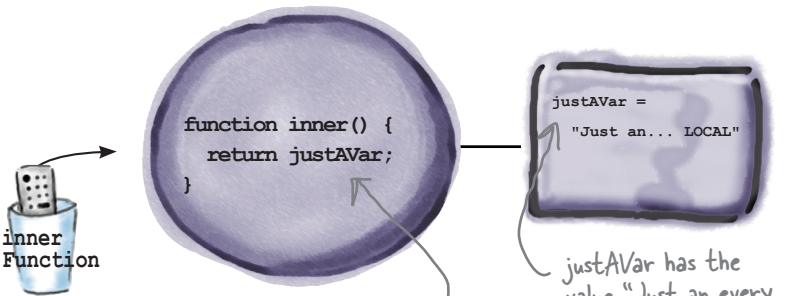
After this statement we have a variable innerFunction that refers to the function (plus an environment) returned from whereAreYou.



---

2 **Next we call innerFunction. To do that we evaluate the code in the function's body, and do that in the context of the function's environment, like this:**

```
var result = innerFunction();
```



3 **Last, we assign the result of the function to the result variable, and then display it in the console.**

```
var result = innerFunction();
console.log(result);
```

innerFunction returns the value "Just an every day LOCAL", which it got from its environment. So, we throw that into the result variable.

And then all we have to do is display that string in the console.

JavaScript console

```
Just an every day LOCAL
```

there are no Dumb Questions

Q: When you say that lexical scope determines where a variable is defined, what do you mean?

A: By lexical scope we mean that JavaScript's rules for scoping are based purely on the structure of your code (not on some dynamic runtime properties). This means you can determine where a variable is defined by simply examining your code's structure. Also remember that in JavaScript only functions introduce new scope. So, given a reference to a variable, look for where that variable is defined in a function from the most nested (where it's used) to the least nested until you find it. And if you can't find it in a function, then it must be global, or undefined.

Q: If a function is nested way down many layers, how does the environment work then?

A: We used a simplistic way of showing the environment to explain it, but you can think of each nested function as having its own little environment with its own variables. Then, what we do is create a chain of the environments of all the nested functions, from inner to outer.

So, when it comes to finding a variable in the environment, you start at the closest one, and then follow the chain until you find your variable. And, if you don't find it, you look in the global environment.

Q: Why are lexical scoping and function environments good things? I would have thought the answer in that code example would be "Oh, don't you worry about it, I'm GLOBAL". That makes sense to me. The real answer seems confusing and counterintuitive.

A: We can see how you might think that, but the advantage of lexical scope is that we can always look at the code to determine the scope that's in place when a variable is defined, and figure out what its value should be from that. And, as we've seen, this is true even if you return a function and invoke it much later in a place totally outside of its original scope.

Now there is another reason you might consider this a good thing, and that is the kind of things we can do in code with this capability. We're going to get to that in just a bit.

Q: Do parameter variables get included in the environment too?

A: Yes. As we've said before, you can consider parameters to be local variables in your functions, so they are included in the environment as well.

Q: Do I need to understand how the environment works in detail?

A: No. What you need to understand is the lexical scoping rules for JavaScript variables, and we've covered that. But now you know that if you have a function that is returned from within a function, it carries its environment around with it.

Remember that JavaScript functions are always evaluated in the same scoping environment in which they were defined. Within a function, if you want to determine where a variable is coming from, search in its enclosing functions, from the most nested to the least.

What the heck is a closure?

Sure, everyone talks about closures as *the must have* language feature, but how many people actually get what they are or how to use them? Darn few. It's the language feature everyone wants to understand and the feature every traditional language wants to add.

Here's the problem. According to many well-educated folks in the business, *closures are hard*. But that's really not a problem for you. Want to know why? No, no, it's not because this is a "brain friendly book" and no, it's not because we have a killer application that needs to be built to teach closures to you. It's because *you just learned them*. We just didn't call them closures.

So without further ado, we give you the super-formal definition.

If you've been trained well in this book you should be thinking at this point, "Ah, this is the 'get a big raise' knowledge."

> **Closure, noun**: A closure is a function together with a referencing environment.

Okay, we agree that definition isn't totally illuminating. But why is it called *closure*? Let's quickly walk through that, because—seriously—this could be one of those make-or-break job interview questions, or the thing that gets you that raise at some point in the future.

To understand the word *closure*, we need to understand the idea of "closing" a function.

Sharpen your pencil

Here's your task: (1) find all the **free variables** in the code below and circle them. A free variable is one that isn't defined in the local scope. (2) Pick one of the environments on the right that **closes the function**. By that we mean that it provides values for all the free variables.

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
       ending = " -- I'm just sayin!";
    } else if (notSoMuch) {
       ending = " -- Not so much.";
    }
    alert(phrase + ending);
}
```

Circle the free variables in this code. Free variables are not defined in the local scope.

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paul";
```

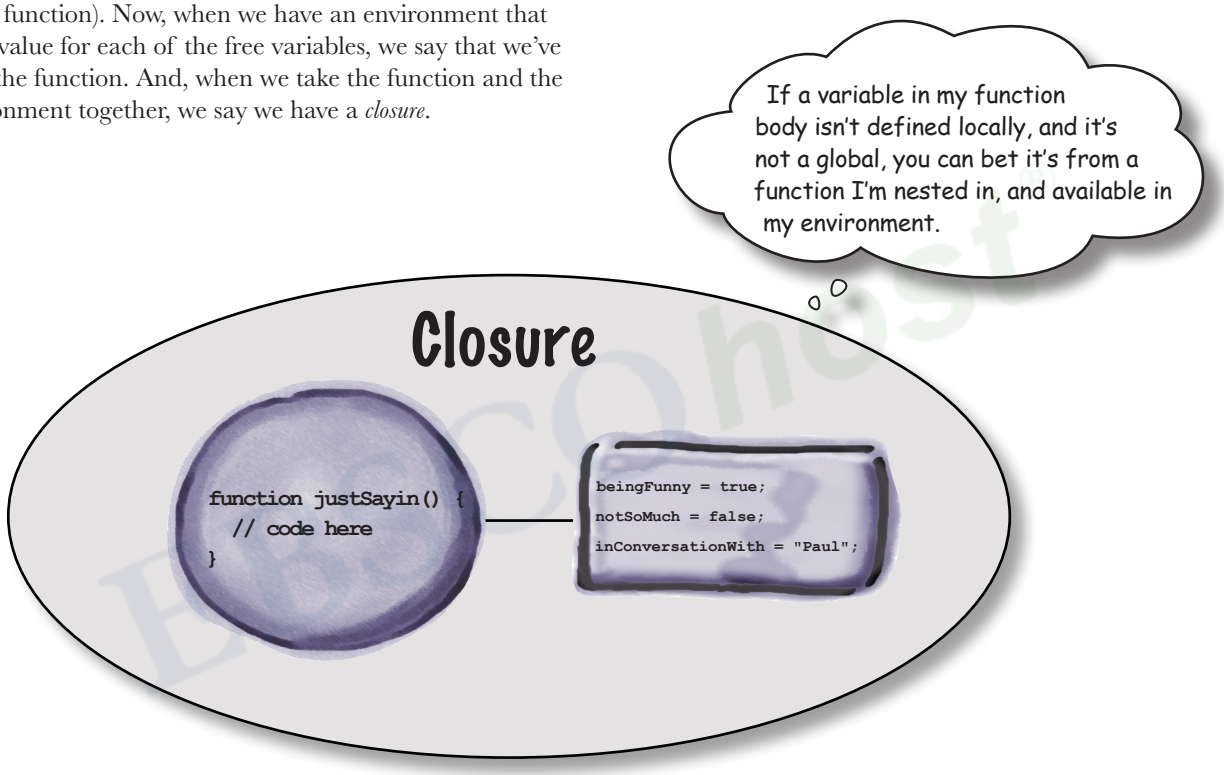
```
beingFunny = true;
justSayin = false;
oocoder = true;
```

```
notSoMuch = true;
phrase = "Do do da";
band = "Police";
```

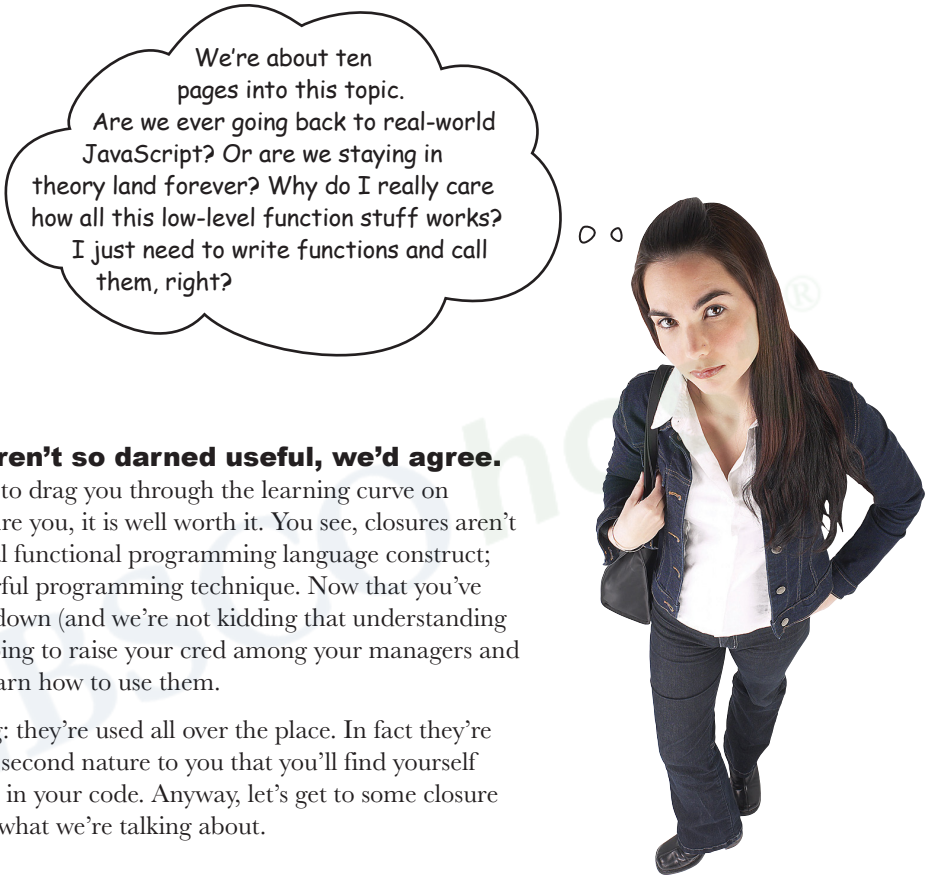
Pick one of these that closes the function.

Closing a function

You probably figured this out in the previous exercise, but let's run through it one more time: a function typically has *local variables* in its code body (including any parameters it has), and it also might have variables that aren't defined locally, which we call *free variables*. The name *free* comes from the fact that within the function body, free variables aren't bound to any values (in other words, they're not declared locally in the function). Now, when we have an environment that has a value for each of the free variables, we say that we've *closed* the function. And, when we take the function and the environment together, we say we have a *closure*.



A closure results when we combine a function that has free variables with an environment that provides variable bindings for all those free variables.



If closures weren't so darned useful, we'd agree.

We're sorry we had to drag you through the learning curve on closures but we assure you, it is well worth it. You see, closures aren't just some theoretical functional programming language construct; they're also a powerful programming technique. Now that you've got how they work down (and we're not kidding that understanding closures is what's going to raise your cred among your managers and peers) it's time to learn how to use them.

And here's the thing: they're used all over the place. In fact they're going to become so second nature to you that you'll find yourself using them liberally in your code. Anyway, let's get to some closure code and you'll see what we're talking about.

Using closures to implement a magic counter

Ever think of implementing a counter function? It usually goes like this:

```
var count = 0;

function counter() {
    count = count + 1;
    return count;
}
```

We have a global variable count.

Each time we call counter, we increment the global count variable, and return the new value.

And we can use our counter like this:

```
console.log(counter());
console.log(counter());
console.log(counter());
```

So we can count and display the value of the counter like this.

JavaScript console

```
1
2
3
```

The only issue with this is that we have to use a global variable for `count`, which can be problematic if you're developing code with a team (because people often use the same names, which end up clashing).

What if we were to tell you there is a way to implement a counter with a totally local and protected `count` variable? That way, you'll have a counter that no other code can ever clash with, and the only way to increment the counter value is through the function (otherwise known as a closure).

To implement this with a closure, we can reuse most of the code above. Watch and be amazed:

```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
```

Here, we're putting the count variable in the function makeCounter. So now count is a local variable, not a global variable.

Now, we create the counter function, which increments the count variable.

And return the counter function.

This is the closure. It holds count in its environment.

Think this magic trick will work? Let's try it and see…

Test drive your magic counter

We added a bit of testing code to test the counter. Give it a try!

```
function makeCounter() {
    var count = 0;

    function counter() {
        count = count + 1;
        return count;
    }
    return counter;
}
var doCount = makeCounter();
console.log(doCount());
console.log(doCount());
console.log(doCount());
```

JavaScript console

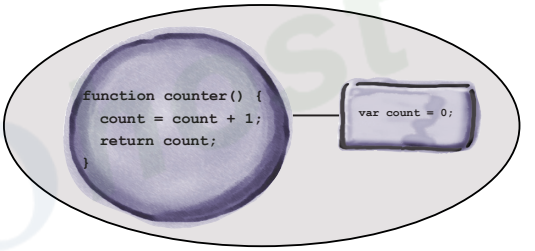
```
1
2
3
```

Our counter works... we get solid counting results.

Looking behind the curtain...

Let's step through the code to see how the counter works.

1. We call makeCounter, which creates a counter function and returns it along with an environment containing the free variable, count. In other words, it creates a closure. The function returned from makeCounter is stored in doCount.
2. We call the function doCount. This executes the body of the counter function.
3. When we encounter the variable count, we look it up in the environment, and retrieve its value. We increment count, save the new value back into the environment, and return that new value to where doCount was called.
4. We repeat steps 2 and 3 each time we call doCount.




```
function makeCounter() {
    var count = 0;

    function counter() {
    3   count = count + 1;
        return count;
    }
    return counter;
}
1 var doCount = makeCounter();
2 console.log(doCount());
  console.log(doCount());
4 console.log(doCount());
```

This is a closure.

When we call doCount (which is a reference to counter) and need to get the value of count, we use the count variable that's in the closure's environment. The outside world (the global scope) never sees the variable count. But we can use it anytime we call doCount. And there's no other way to get to count except by calling doCount.

When we call makeCounter, we get back a closure: a function with an environment.

It's your turn. Try creating the following closures. We realize this is not an easy task at first, so refer to the answer if you need to. The important thing is to work your way through these examples, and get to the point where you fully understand them.

First up for 10pts: makePassword takes a password as an argument and returns a function that accepts a password guess and returns true if the guess matches the password (sometimes you need to read these closure descriptions a few times to get them):

```
function makePassword(password) {
    return _____________________________ {
        return (passwordGuess === password);
    };
}
```

Next up for 20pts: the multN function takes a number (call it n) and returns a function. That function itself takes a number, multiplies it by n and returns the result.

```
function multN(n) {
    return _____________________________ {
        return _______________;
    };
}
```

Last up for 30 pts: This is a modification of the counter we just created. makeCounter takes no arguments, but defines a count variable. It then creates and returns an object with one method, increment. This method increments the count variable and returns it.

Creating a closure by passing a function expression as an argument

Returning a function from a function isn't the only way to create a closure. You create a closure *whenever* you have a reference to a function that has free variables, and that function is executed outside of the context in which it was created.

Another way we can create a closure is to pass a function to a function. The function we pass will be executed in a completely different context than the one in which it was defined. Here's an example:

```
function makeTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

}

makeTimer("Cookies are done!", 1000);
```

We have a function...

...with a free variable...

...that we are using as a handler for setTimeout.

...and this function will be executed 1000 milliseconds from now, long after the function makeTimer has completed.

Here, we're passing a function expression that contains a free variable, `doneMessage`, to the function `setTimeout`. As you know, what happens is we evaluate the function expression to get a function reference, which is then passed to `setTimeout`. The `setTimeout` method holds on to this function (which is a function plus an environment—in other words, a closure) and then 1000 milliseconds later it calls that function.

And again, the function we're passing into `setTimeout` is a closure because it comes along with an environment that binds the free variable, `doneMessage`, to the string "Cookies are done!".

BRAIN POWER

What would happen if our code looked like this instead?

```
function handler() {
    alert(doneMessage);
}
function makeTimer(doneMessage, n) {
    setTimeout(handler, n);
}
makeTimer("Cookies are done!", 1000);
```

BRAIN 2 POWER

Revisit the code on page 412 in Chapter 9. Can you modify your code to use a closure, and eliminate the need for the third argument to setTimeout?

The closure contains the actual environment, not a copy

One thing that often misleads people learning closures is that they think the environment in the closure must have a copy of all the variables and their values. It doesn't. In fact, the environment references the live variables being used by your code, so if a value is changed by code outside your closure function, that new value is seen by your closure function when it is evaluated.

Let's modify our example to see what that means.

```
function setTimer(doneMessage, n) {

    setTimeout(function() {
        alert(doneMessage);
    }, n);

    doneMessage = "OUCH!";
}
setTimer("Cookies are done!", 1000);
```

The closure is created here.

Now we're changing the value of doneMessage after we call setTimeout.

1. When we call setTimeout and pass to it the function expression, a closure is created containing the function along with a reference to the environment.

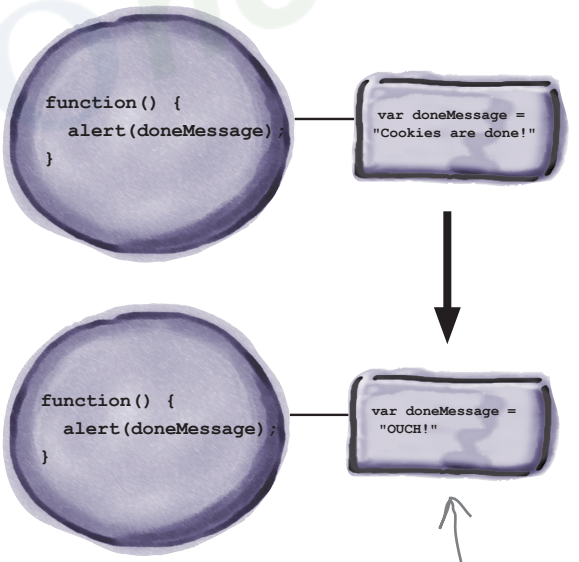
```
setTimeout(function() {
    alert(doneMessage);
}, n);
```

2. Then, when we change the value of doneMessage to "OUCH!" outside of the closure, it's changed in the same environment that is used by the closure.

```
doneMessage = "OUCH!";
```

3. 1000 milliseconds later, the function in the closure is called. This function references the doneMessage variable, which is now set to "OUCH!" in the environment, so we see "OUCH!" in the alert.

```
function() { alert(doneMessage); }
```



When the function is called, it uses the value for doneMessage that's in the environment, which is the new value we set it to earlier, in setTimer.

Creating a closure with an event handler

Let's look at one more way to create a closure. We'll create a closure with an event handler, which is something you'll see fairly often in JavaScript code. We'll start by creating a simple web page with a button and a `<div>` element to hold a message. We'll keep track of how many times you click the button and display the tally in the `<div>`.

Here's the HTML and a tiny bit of CSS to create the page. Go ahead and add the HTML and CSS below into a file named "divClosure.html".

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    // JavaScript code here
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>
```

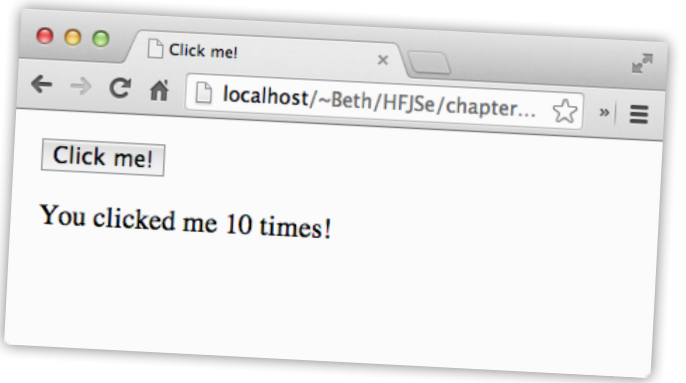
Just your typical, basic web page...

With a little CSS to style the elements in the page.

Here's where our code's going to go.

We have a button, and a to hold the message we'll update each time you click the button.

Here's what we're going for: each time you click the button, the message in the will be updated to show the number of times you've clicked.



Next, let's write the code. Now, you could write the code for this example without using a closure at all, but as you'll see, by using a closure, our code is more concise, and even a bit more efficient.

Click me! without a closure

Let's first take a look at how you'd implement this example *without* a closure.

```
var count = 0;

window.onload = function() {
    var button = document.getElementById("clickme");
    button.onclick = handleClick;
};

function handleClick() {
    var message = "You clicked me ";
    var div = document.getElementById("message");
    count++;
    div.innerHTML = message + count + " times!";
}
```

The count variable will need to be a global variable, because if it's local to handleClick (the click event handler on the button, see below), it'll just get re-initialized every time we click.

In the load event handler function, we get the button element, and add a click handler to the onclick property.

Here's the button's click handler function.

We define the message variable...

...get the element from the page...

...increment the counter...

...and update the with the message containing how many times we've clicked.

Click me! with a closure

The version without a closure looks perfectly reasonable, except for that global variable which could potentially cause trouble. Let's rewrite the code using a closure and see how it compares. We'll show the code here, and take a closer look after we test it.

```
window.onload = function() {
    var count = 0;
    var message = "You clicked me ";
    var div = document.getElementById("message");

    var button = document.getElementById("clickme");
    button.onclick = function() {
        count++;
        div.innerHTML = message + count + " times!";
    };
};
```

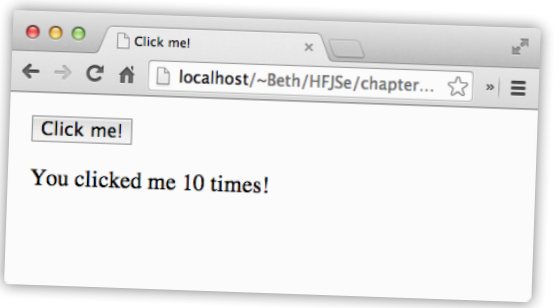
Now, all our variables are local to window.onload. No problems with name clashing now.

We're setting up the click handler as a function expression assigned to the button's onclick property, so we can reference div, message and count in the function. (Remember your lexical scoping!)

This function has three free variables: div, message and count, so a closure is created for the click handler function. So what gets assigned to the button's onclick property is a closure.

Test drive your button counter

Okay, let's bring the HTML and the code together in your "divClosure.html" file and give this a test run. Go ahead and load the page and then click on the button to increment the counter. You should see the message update in the . Look at the code again, and make sure you think you know how this all works. After you've done so, turn the page and we'll walk through it together.



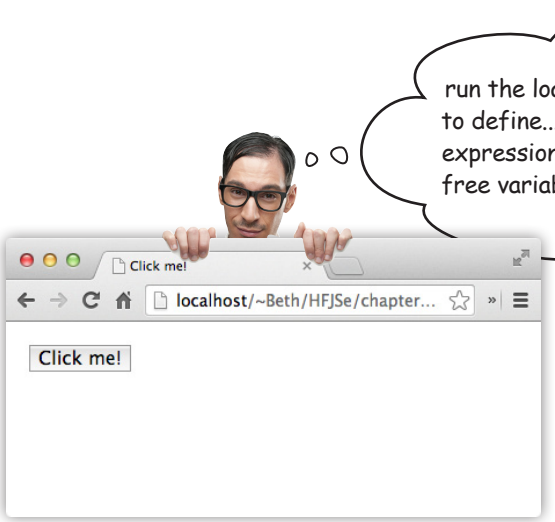
Here's what we got.

Update your "divClosure.html" file like this.

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Click me!</title>
<style>
    body, button { margin: 10px; }
    div { padding: 10px; }
</style>
<script>
    window.onload = function() {
        var count = 0;
        var message = "You clicked me ";
        var div = document.getElementById("message");

        var button = document.getElementById("clickme");
        button.onclick = function() {
            count++;
            div.innerHTML = message + count + " times!";
        };
    };
</script>
</head>
<body>
    <button id="clickme">Click me!</button>
    <div id="message"></div>
</body>
</html>
```

How the Click me! closure works

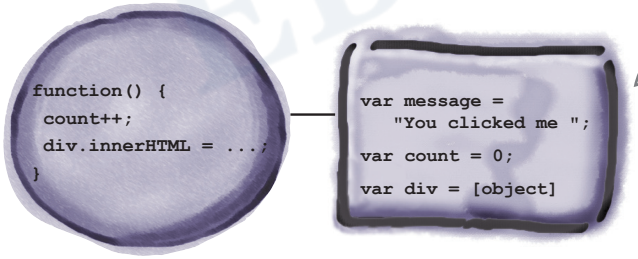
To understand how the closure works, let's follow along with the browser once again, as it evaluates this code…

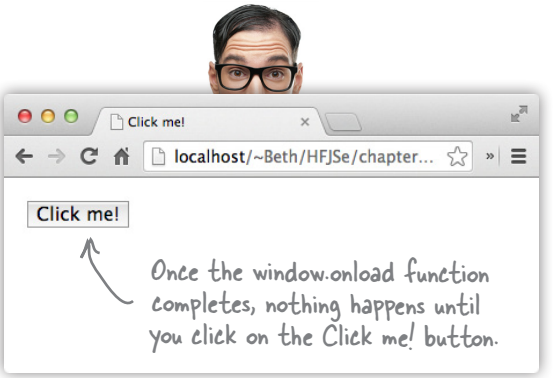


The page has loaded so I can run the load handler. I've got some variables to define... Oh, and I see I've got a function expression. Let's see... that's referencing three free variables, so I'd better make a closure.

```
window.onload = function() {
   var count = 0;
   var message = "You clicked me ";
   var div = document.getElementById("message");

   var button = document.getElementById("clickme");
   button.onclick = function() {
      count++;
      div.innerHTML = message + count + " times!";
   };
};
```

The browser creates a closure for the function that's assigned to the button.onclick property. The environment has the div variable, the message variable and the count variable.

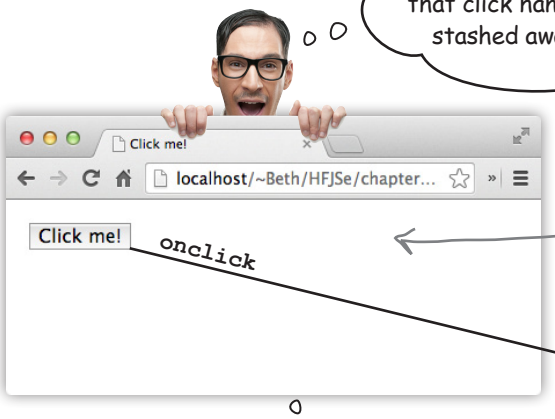




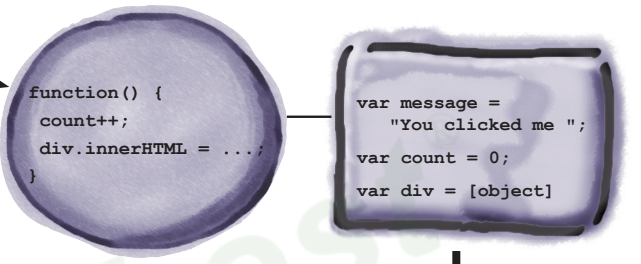
Once the window.onload function completes, nothing happens until you click on the Click me! button.

And now I'll assign that closure to the onclick property of the Click me! button in the page. Okay done with the load handler... now I've just got to wait for someone to click the button.

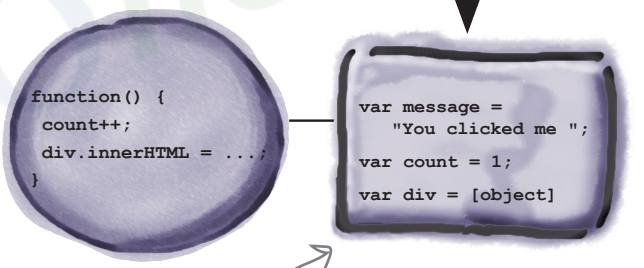
Hey, someone clicked the button! Time to execute that click handler function I stashed away earlier...



Even though the button variable doesn't exist anymore (it goes away when the window.onload function has completed), the button object is in the DOM, and it has our closure stored in its onclick property.



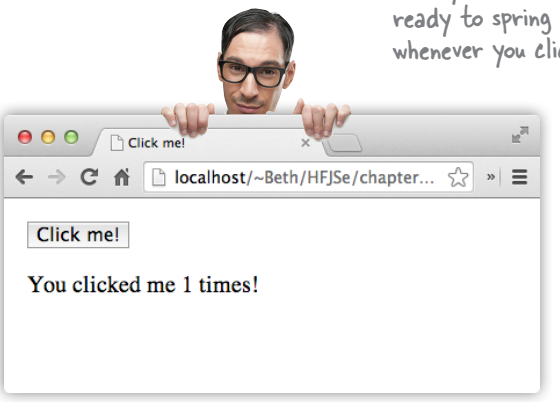
Oh, I see we have a closure. Good, that means I can find values for the free variables in the environment.



Notice that the div variable in the closure holds an object. When div was initialized in window.onload we stored the object returned by document.getElementById in the div variable, so we don't have to get the object from the DOM again; we have it already. This saves us a little bit of computation and makes our code just a tiny bit faster.

The closure doesn't go away until you close the page. It's ready to spring into action whenever you click the button!

I've incremented the count variable by one and made sure the value is updated in the environment. And I've updated the message in the page, so now all I have to do is wait for another click.



REVISITED

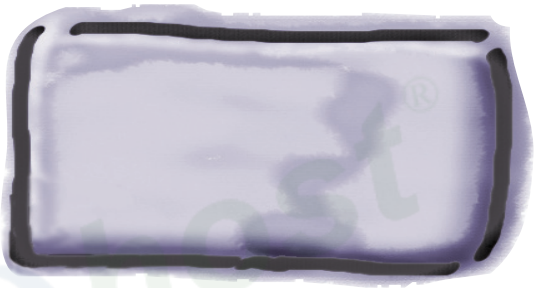
EXTREME JAVASCRIPT CHALLENGE

We need a closures expert and we've heard that's you! Now you know how closures work, can you figure out why both specimens below evaluate to 008? To figure it out, write any variables that are captured in the environments for the functions below. Note that it's perfectly fine for an environment to be empty. Check your answer at the end of the chapter.

Specimen #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

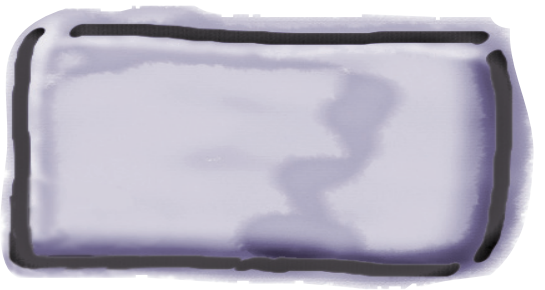
Environment



Specimen #2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

Environment



Sharpen your pencil

First, check out this code:

```
(function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
})("cookies");
```

Using a function expression in place of a reference, taken to the extreme.

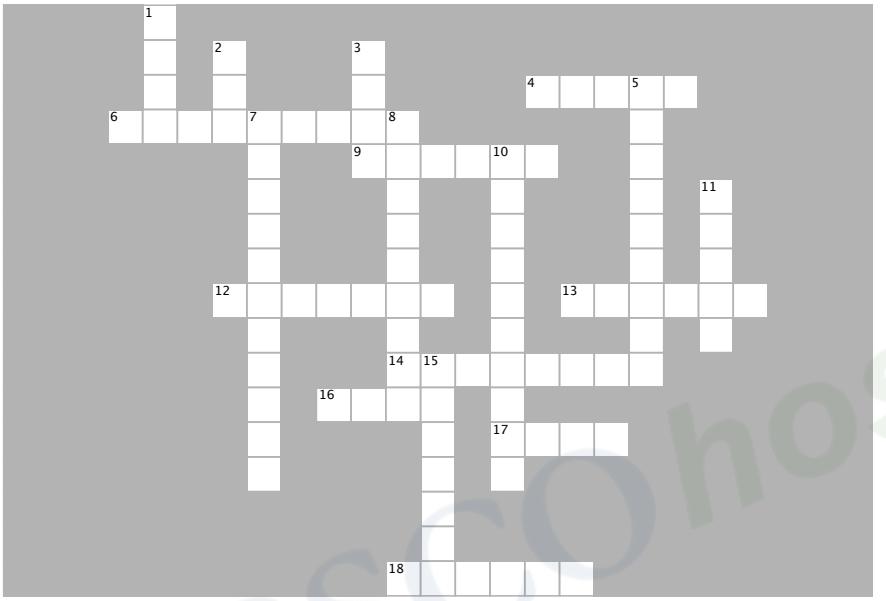
Your task is to figure out not just what this code computes, but *how* it computes. To do that, go in reverse. That is, take out the anonymous function, assign it to a variable, and then use that variable where the function expression used to be. Is the code more obvious now? So, what does it do?

BULLET POINTS

- An **anonymous function** is a function expression that has no name.
- Anonymous functions can make your code more concise.
- A **function declaration** is defined before the rest of your code is evaluated.
- A **function expression** is evaluated at runtime with the rest of your code, and so is not defined until the statement in which it appears is evaluated.
- You can pass a function expression to another function, or return a function expression from a function.
- A function expression evaluates to a **function reference**, so you can use a function expression anywhere you can use a function reference.
- **Nested functions** are functions defined inside another function.
- A nested function has local scope, just like other local variables.
- **Lexical scope** means that we can determine the scope of a variable by reading our code.
- To bind the value of a variable in a nested function, use the value that's defined in the closest enclosing function. If no value is found, then look in the global scope.
- **Closures** are a function along with a referencing environment.
- A closure captures the value of variables in scope at the time the closure is created.
- **Free variables** in the body of a function are variables that are not bound in the body of that function.
- If you execute a function closure in a different context in which it was created, the values of free variables are determined by the referencing environment.
- Closures are often used to capture state for event handlers.

JavaScript cross

Time for another crossword puzzle to burn some JavaScript into those neuron pathways.



ACROSS

4. A function declaration nested in another function has _____ scope.

6. When we tried to call fly before it was defined, we got this kind of error.

9. wingFlapper is a _____ function.

12. We often use setTimeout to create a timer for making _____.

13. A function expression assigned to a variable at the top level of your code has _____ scope.

14. To get a raise, you should understand how _____ work.

16. A _____ variable is one that's not defined in the local scope.

17. We changed the value of doneMessage to _____ in the closure.

18. An environment that provides values for all free variables _____ a function.

DOWN

1. _____ is always right.

2. ____ changed his shirt between pages.

3. Movie the word "derezzed" was used in.

5. An _____ function is a function expression that has no name.

7. A function with an _____ attached to it is called a closure.

8. A function expression evaluates to a function _____.

10. We passed a function _____ to set the cookie alarm.

11. Parameters are _____ variables, so they're included in the environment where variables are defined.

15. _____ scope means you can understand the scope of your variables by reading the structure of your code.

Sharpen your pencil Solution

There are a few opportunities in the code below to make the code more concise by using anonymous functions. Go ahead and rework the code to use anonymous functions wherever possible. You can scratch out the old code and write in new code where needed. Oh, and one more task: circle any anonymous functions that are already being used in the code. Here's our solution.

```
window.onload = init;
var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
        console.log("Baking the cookies.");
        setTimeout(done, time);
    }
};
function init() {
    var button = document.getElementById("bake");
    button.onclick = handleButton;
}
function handleButton() {
    console.log("Time to bake the cookies.");
    cookies.bake(2500);
}
function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

We reworked the code to create two anonymous function expressions, one for the init function, and one for the handleButton function.

Now we assign a function expression to the window.onload property...

```
window.onload = function() {
    var button = document.getElementById("bake");
    button.onclick = function() {
        console.log("Time to bake the cookies.");
        cookies.bake(2500);
    };
};

var cookies = {
    instructions: "Preheat oven to 350...",
    bake: function(time) {
            console.log("Baking the cookies.");
            setTimeout(done, time);
        }
};

function done() {
    alert("Cookies are ready, take them out to cool.");
    console.log("Cooling the cookies.");
    var cool = function() {
        alert("Cookies are cool, time to eat!");
    };
    setTimeout(cool, 1000);
}
```

...and assign a function expression to the button.onclick property.

Extra credit for you if you figured out you can pass the cool function directly to setTimeout, like this:

```
setTimeout(function() {
    alert("Cookies are cool, time to eat!");
}, 1000);
```

Let's make sure you have the syntax down for passing anonymous function expressions to other functions. Convert this code from one that uses a variable (in this case vaccine) as a parameter to one that uses an anonymous function. Here's our solution.

```
administer(patient, function(dosage) {
    if (dosage > 0) {
        inject(dosage);
    }
}, time);
```

Notice that it's totally fine to use more than one line for a function expression that's used as an argument. But watch your syntax; it's easy to make a mistake!

It's your turn. Try creating the following closures. We realize this is not an easy task at first, so refer to the answer if you need to. The important thing is to work your way through these examples, and get to the point where you fully understand them.

Here are our solutions:

First up for 10pts: makePassword takes a password as an argument and returns a function that accepts a password guess and returns true if the guess matches the password (sometimes you need to read these closure descriptions a few times to get them):

```
makePassword(password) {
    return function guess(passwordGuess) {
        return (passwordGuess === password);
    };
}
var tryGuess = makePassword("secret");
console.log("Guessing 'nope': " + tryGuess("nope"));
console.log("Guessing 'secret': " + tryGuess("secret"));
```

The function that's returned from makePassword is a closure with an environment containing the free variable password.

We pass in the value "secret" to makePassword, so this is the value that's stored in the closure's environment.

Notice here we're using a named function expression! We don't have to, but it's handy as a way to refer to the name of the inner function. But also notice we must invoke the returned function using tryGuess (not guess).

And when we invoke tryGuess, we compare the word we pass in ("nope" or "secret") with the value for password in the environment for tryGuess.

The solutions continue on the next page...

It's your turn. Try creating the following closures. We realize this is not an easy task at first, so refer to the answer if you need to. The important thing is to work your way through these examples, and get to the point where you fully understand them.

Here are our solutions (continued):

Next up for 20pts: the multN function takes a number (call it n) and returns a function. That function itself takes a number, multiplies it by n and returns the result.

```
function multN(n) {
    return function multBy(m) {
        return n*m;
    };
}
var multBy3 = multN(3);
console.log("Multiplying 2: " + multBy3(2));
console.log("Multiplying 3: " + multBy3(3));
```

The function that's returned from multN is a closure with an environment containing the free variable n.

So we invoke multN(3) and get back a function that multiplies any number you give it by 3.

Last up for 30 pts: This is a modification of the counter we just created. makeCounter takes no arguments, but defines a count variable. It then creates and returns an object with one method, increment. This method increments the count variable and returns it.

```
function makeCounter() {
    var count = 0;
    return {
        increment: function() {
            count++;
            return count;
        }
    };
}
var counter = makeCounter();
console.log(counter.increment());
console.log(counter.increment());
console.log(counter.increment());
```

This is similar to our previous makeCounter function, except now we're returning an object with an increment method, instead of returning a function directly.

The increment method has a free variable, count. So, increment is a closure with an environment containing the variable count.

Now, we call makeCounter and get back an object with a method (that is a closure).

We invoke the method in the usual way, and when we do, the method references the variable count in its environment.

Sharpen your pencil Solution

Use your knowledge of functions and variables and check off the true statements below. Here's our solution:

- [x] The handler variable holds a function reference.
- [x] When we assign handler to window.onload, we're assigning it a function reference.
- [x] The only reason the handler variable exists is to assign it to window.onload.
- [x] We'll never use handler again as it's code that is meant to run only when the page first loads.
- [x] Invoking onload handlers twice is not a great idea—doing so could cause issues given these handlers usually do some initialization for the entire page.
- [x] Function expressions create function references.
- [x] Did we mention that when we assign handler to window.onload, we're assigning it a function reference?

Sharpen your pencil Solution

Here's your task: (1) find all the **free variables** in the code below and circle them. A free variable is one that isn't defined in the local scope. (2) Pick one of the environments on the right that **closes the function**. By that we mean that it provides values for all the free variables. Here's our solution.

```
function justSayin(phrase) {
    var ending = "";
    if (beingFunny) {
       ending = " -- I'm just sayin!";
    } else if (notSoMuch) {
       ending = " -- Not so much.";
    }
    alert(phrase + ending);
}
```

Circle the free variables in this code. Free variables are not defined in the local scope.

This environment closes the two free variables beingFunny and notSoMuch

```
beingFunny = true;
notSoMuch = false;
inConversationWith = "Paul";
```

```
beingFunny = true;
justSayin = false;
oocoder = true;
```

```
notSoMuch = true;
phrase = "Do do da";
band = "Police";
```

Pick one of these that closes the function.

SOLUTION

EXTREME JAVASCRIPT CHALLENGE

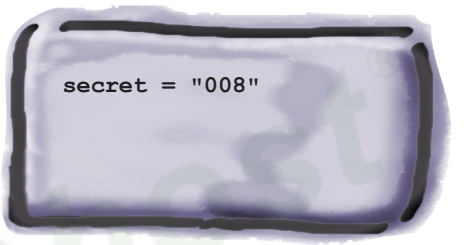
We need a closures expert and we've heard that's you! Now you know how closures work, can you figure out why both specimens below evaluate to 008? To figure it out, write any variables that are captured in the environments for the functions below. Note that it's perfectly fine for an environment to be empty. Here's our solution.

Specimen #1

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue();
}
getSecret();
```

secret is a free variable in getValue...

Environment



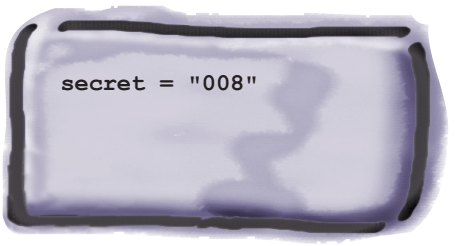
...so it's captured in the environment for getValue. But we don't return getValue from getSecret, so we never see the closure outside the context in which it was created.

Specimen #2

```
var secret = "007";

function getSecret() {
    var secret = "008";

    function getValue() {
        return secret;
    }
    return getValue;
}
var getValueFun = getSecret();
getValueFun();
```

secret is a free variable in getValue...

Environment



...and here, we do create a closure that's returned from getSecret. So when we invoke getValueFun (getValue) in a different context (the global scope), we use the value of secret in the environment.

Sharpen your pencil Solution

Here's our solution for this brain twister!

```
(function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
})("cookies");
```

Your task is to figure out not just what it computes, but how it computes. To do that, go in reverse, that is, take out the anonymous function, assign it to a variable, and then replace the previous function with the variable. Is the code more obvious now? So what does it do?

Here's the function, extracted. We just called it eat. You could have made this a function declaration if you preferred.

```
var eat = function(food) {
    if (food === "cookies") {
        alert("More please");
    } else if (food === "cake") {
        alert("Yum yum");
    }
};
(eat)("cookies");
```

You would write this as eat("cookies") of course, but we're showing how to substitute eat for the function expression above.

And what we're doing is calling eat on "cookies". But what are the extra parentheses for?

So all this code did was to inline a function expression and then immediately invoke it with some arguments.

Oh, and it returns "More please".

Here's the deal. Remember how a function declaration starts with the word function followed by a name? And remember how a function expression needs to be inside a statement? Well, if you don't use parentheses around the function expression, the JavaScript interpreter wants this to be a declaration rather than a function expression. But we don't need the parentheses to call eat, so you can remove them.

JavaScript cross Solution