How Scope Works in Functions

In this lesson we're going to talk a little bit about **closures**, and how they work.

In order to get a grasp on how closures work, I'm going to give you a quick refresher on how scope works in functions. So variables that are defined within the scope of a function, only exist until that function terminates, and that's easy to demonstrate.

```
var publicFunction = function() {
  var privateVariable = "something";
  return privateVariable;
};
//console.log(privateVariable); //error

var privateValue = publicFunction();
console.log(privateValue); //"something"
```

We can't access `privateVariable` as it was defined inside of `publicFunction`, so we can't access it outside of this function. But functions can definitely return values, and primitive types, such as the string that we've defined as `privateVariable`, would be passed by value.

Functions are reference types, and that means that they're passed or returned from a function by reference to one shared original instead of being passed by value the way that a string is.

Creating innerFunction

We can modify our example by adding in another variable inside the function:

```
var publicFunction = function() {
  var privateVariable = "something";
  var innerFunction = function() {
    return privateVariable;
  };
  return innerFunction;
};
```

```
var privateFunction = publicFunction();
console.log(privateFunction()); //"something"
```

When we created `innerFunction` inside of `publicFunction`, it had to have access to all of the variables that were in scope where it was created, which includes something. When we accessed that outside of `privateFunction` it still had access to that “something”. The `innerFunction` that we passed, retained access, it enclosed that `privateVariable` inside a `publicFunction`. And that created a closure around that variable, and everything else that was in scope when inner function was defined.