JavaScript: Functions

1. Functions

1.1 What are functions?

A function is just a series of statements that are grouped together into a special package in JavaScript.

We're going to explore the anatomy of a function:

- all functions are defined with the function keyword, regardless of how they are used or invoked. Defining the function is sometimes called declaring the function.

- Before they can be used, all functions have to be declared or defined. it makes the function available to your program.

- if you used to traditional functions, this might seem weird but in JavaScript, the name of the function is sometimes optional.

- when you're creating a function name you can use a dollar sign $, _ an underscore, letters A...Z, the numbers 0…9 and a number of other special characters. A function like a variable, however, cannot start with a number.

- After the function name, you have a set of parentheses, and inside those, you can enter a series of parameters separated by commas. You can have zero or as many parameters as you want, and these will become variables that are local to and only available inside your function.

- A function include a series of instructions. We call those statements and they go inside curly braces. They're usually separated by semicolons.

```
function plus(a, b) {
  var sum = a+b;
  return sum;
}

console.log(plus(2,2));
```

- JavaScript provides a special statement called return. In our example, it will return the value of our addition. What makes JavaScript special is that a function can really return anything. JavaScript can return an object or even another function. That lets JavaScript do things that are very difficult to do in other languages.

- In order to use the function, you'll need to call it within your code. This is called invoking the function.

1.2 Declaring Functions

The two ways of declaring functions. There’s another way to create a function called a definition expression. The difference between function declarations and definition expressions is how they're invoked.

- The other way to create a function is through something called a definition expression. You can assign a function as the value of a variable or an expression. This is sometimes called a function literal.

- It's also known as an anonymous function. It’s called an anonymous function because you don't need to provide a name for it.

- If you don't provide a name for it, though, it's not bound to any identifier. So sometimes, the names can be a little bit useful especially in patterns like recursion.

- There's a lot that you can do with a function literal that you can't do with a function declaration, and that's because you can use it anywhere that a variable will go.

```
 var plus = function(a,b) {
  return console.log(a+b);
};

plus(2,2);
```

- You can invoke the function immediately. We can actually ask JavaScript to execute this function immediately and all we would have to do is put the parentheses and initialize the value immediately by passing along some parameters. This is called initializing or instantiating the function.

- A definition expression is very useful when a function is only needed once, or if you want to execute something immediately. So the different ways of declaring functions are going to give us some powerful ways of handling our code blocks.

```
var plus = function(a,b) {
  return console.log(a+b);
}(2, 3);          //Returns 5.
```

Although function declarations are more common in other languages, definition expressions are handled differently in JavaScript, and it's part of what makes it so powerful.

1.3 Invoking Functions Traditionally

Function code is not executed until that function is called or invoked. We use the term invoking because some functions have no name and you can't really say you are calling something that doesn't have a name.

- There's four ways of invoking functions. You can invoke them as functions, as methods, as constructors, and through the call and apply methods. The first two are definitely the most common, and the ones that you may have seen already in some programs.

- Functions also get a couple of extra parameters: ***arguments*** and ***this***.

The ***arguments*** parameter is useful when we want to create functions without passing a specific number of parameters. So in our example, when we want to add two numbers, we created a function that had a and b as the variables. What if you wanted to have three numbers or an infinite amount of numbers? You can't really create a certain amount of variables

if you don't know how many there are. But the arguments array lets you do that.

The ***this*** parameter is a little bit trickier. And the way we call a function (the four different ways) has an effect on the value of the *this* argument. This is especially difficult to talk about because it's hard to not say this when you are trying to explain the argument *this*.

- Another property of functions is that invoking them passes control of our program to the function that we're calling.

- If the function has a name you can call it by it's name and then pass along zero or more parameters. This way of calling a function is the traditional invocation method.

- The interesting thing about traditional invocation is what happens to the *this* parameter. With traditional invocation the *this* parameter gets a copy of the global object. And that can be a little bit weird, bad, and create some problems.

Example: I'm going create a function again. And I'll use the return statement. I'm going to return a series of things. So I'm going to use the return method as a function and as a matter of fact, the return method is a function and therefore you can use it as you would a function. So we can actually pass it along a series of parameters. The first one we'll pass is the console log a plus b which is the result of our operation, then stick a comma. Just like I would with a function. Notice that when I call a function, I usually pass it along some parameters. I'm doing the same thing to this return function. This is a straight function call.

```
console.log(this)
```

Returns the window object. this is odd because you may have been expecting to get something related to this function. But as I mentioned, invoking the function in this way makes the this argument have the global object, which is the window object. So this is almost like having the entire browser available to you. When you're programming with functions that can be a little bit tricky because you may not want that big humongous global argument. You may want something related to that function.

```
console.log(arguments)
```

Returns an array-like object. I say array-like object because it's not exactly an array but it looks just like one. And right now it's giving you a list of the two parameters that we pass the function.

```
function plus(a, b) {
 return (
   console.log(a+b),              //Returns 4.
   console.log(this),             //Returns the Window object.
   console.log(arguments)         //Returns [2, 2] .
 )
}

plus(2,2)
```

So hopefully by learning how to invoke functions, you learned a little bit more about how they worked. You learned about the four different ways of invoking functions and how the traditional methods affect the *this* parameter.

2. Function Invocation

2.1 Using Functions as Objects

In addition to the traditional way of calling functions, we can also invoke them as methods. A method is nothing more than a function that has been assigned as a property of an object.

- Before we talk about this invocation method, let's discuss how objects work. what's an object? An object is nothing more than one of JavaScript's many data types. In JavaScript, for example, you can have variables. A variable will let you hold regular values, but you can also have lists, which are also known as arrays. Arrays let you hold lists of elements separated by commas. Those elements can be variables, they can be strings, they can be numbers or Booleans. Objects are the most flexible data type, because they can hold any other data type, including other objects and functions.

- So here's what a typical object looks like. Objects always start and end with curly braces. Inside we have a number of properties, and they are separated by name and value pairs. Then we have a property that is an array, it has the brackets at the beginning and end, but each element in the array is a set of objects, once again. And so, they have curly braces at the beginning and end, and name and property pairs.

```
plus: function (a,b) {
```

Assign functions as the property. Function literal, or a function without a name, also called an anonymous function. Because we don't really need a name for it. We're already going to be calling with the (plus) identifier.

Note: even though I've created the object and the function, the *this* attribute is not get instantiated until I make a call to the function.

```
console.log(this)
```

Instead of returning the global object, it's returning the object that has this function. It’s returning all the parameters of that object, which are the status, as well as the function we call plus.

```
console.log(this.status)
```

By using the *this* attribute, I was able to query the parameter status of the object. So this is one of the big advantages of calling functions as methods. The *this* attribute gets the value of the object, which is really useful.

So let's review.

- The *this* argument is going to point to the object that the function is in.

- You can invoke the function using dot notion.

- The binding of *this* is only going to happen at invocation time. So the *this* attribute is not going to be bound to that object until we invoke the method. Now that actually makes the function highly reusable. So binding functions to objects and using them as methods is a great programming technique. You've probably already used it in all of your JavaScript applications. It's a lot more flexible and useful than any of the other methods. The *this* argument now becomes much more powerful and relevant than before.

```
var calc = {
  status: 'Awesome',
  plus: function (a,b) {
    return (
      console.log(this),       // Returns the object that has this function.
      console.log(a+b),              //Returns the result of the addition 4.
      console.log(arguments),      //Returns the arguments [2, 2] .
      console.log(this.status)       //Returns the word Awesome.
    )
  }
}

calc.plus(2,2);
```

2.2 Invoking Instances through the Constructor

In the last lesson, we created objects manually by first assigning them to a variable and then adding properties and methods.

- Functions in JavaScript can do more than just help you create methods for existing objects. They can create new objects themselves, which is called constructing an object.

- You create an object with the *new* keyword. This method of creating an object is called a constructor invocation. It can use the function keyword to create a copy of an object with the variables from the function as its properties.

```
var Dog = function() {
```

Now this dog is essentially an object and we create a new instance of this object with the new constructor.

```
firstDog = new Dog;
firstDog.name = "Rover";
firstDog.breed = "Doberman";
```

So we could do firstDog and say that we want to create a new object based on this dog function. Once we have that new copy we can assign a name and a breed.

```
console.dir(firstDog);
```

Returns the dog and if you open it you'll notice we have a new instance. And this instance has a breed of Doberman and the name of Rover.

- The *new* keyword created a new instance, which is sort of like a copy but not really. It's more like reproduction. The constructor creates an object that is based on the original function.

- Each instance is going to have its own set of properties. Doing console.dir for the secondDog, I'm going to see that I have two dogs but

each instance of these dogs is different. So they're sort of like copies but not exactly.

```
return console.dir(this);
```

The *this* argument is going to point to each instance of the object. If we called it within an object, it will be referring to the object. Otherwise, the *this* attribute would be referring to the global object.

Whenever we use the constructor operator new, the *this* parameter is going to contain a copy of the object that we've created.

- Convention says that constructor names should be capitalized. In my function, I capitalize the name dog. This is sort of convention and it's a good way of letting developers, and yourself, later on know that this function is a little bit special because it creates an object.

A lot of times generating an object by adding a method like we did in the last movie is the right way to create objects. However, the object constructor allows you to invoke new instances of objects dynamically. This is JavaScript's attempt at being somewhat object oriented.

```
var Dog = function() {
  var name, breed;
  return console.dir(this);
}

firstDog = new Dog;
firstDog.name = "Rover";
firstDog.breed = "Doberman";

secondDog = new Dog;
secondDog.name = "Fluffy";
secondDog.breed = "Poodle"
```

2.3 Expanding Functionality through Prototype

So in the last lesson, we saw how we can create our own objects using functions. The functions themselves become a constructor. What if we wanted to expand the functionality of a constructor by adding say a method?
- We can do that through that constructor's prototype object. So what is this prototype object? Well, JavaScript is known as a prototypal inheritance language. That means that you can base the functionality of an object on another object.
- Every object can be based on another object, and as a matter of fact, every object in JavaScript is based on a different object. That makes it kind of convenient because we don't have to keep on building the same functionality for different things.
- The way that we do this is by linking an object's prototype object to another. This is going to make more sense with an example.

Recreate the Dog function again. To add some functionality to my Dog object. I could do it in the constructor function, but what if I wanted to use that same functionality for something else. So, say that I wanted to create a method called Speak. And I wanted to have different types of animals use the same functionality.

```
Dog.prototype.speak = speak;
```

to expand on my dog by accessing its prototype and then setting a method of the dog To be the same as my speak function. it gives the dog the ability to speak, or the method called speak.

In the console typing firstDog returns:
Dog {name: "Rover", breed: "Doberman", speak: function}
So we can look at our first dog instance, and notice that it says it has a name, a breed, and it also inherited this function, called speak.

```
Cat.prototype.speak = speak;
```

Multiple objects can inherit the same functionality. So now that we have this speaking functionality, we can add it to other objects. And now I get the two different animals saying things.

Now each of them are instances. They have the same functionality. So now, that allows me to modify one simple function to control the functionality of two different objects.

- In JavaScript all objects inherit properties. And as a matter of fact, any instances or any functions that we create as function declarations inherit from the *function* object.
- *function* itself is a constructor and it also inherits from *object*.

As a matter of fact, if I were to define a variable and set it up to an empty object. I would see that this is an object.

In the console:
```
var myThing = {};
console.dir(myThing)
```
Returns: Object

Now if I do a console directory on the speak function, I'll notice that it's defined as a function. But if I look into these, you'll see that, that function is using as its prototype, the *object*. So this function is actually based on the object constructor.

In the console:
```
console.dir(speak)
```
Returns:
```
function (saywhat) {  console.log(saywhat);  }
    arguments: null
    caller: null
    length: 1
    name: ""
    prototype: speak            //RayV get Object instead of speak.
    __proto__: function Empty() {}
    <function scope>
```

This ability to create relationships in JavaScript is really valuable. With the prototype object, we can expand the functionality of anything in JavaScript. And yes, that includes functions, objects, and even the master constructors themselves.
So the possibility for taking any existing functionality within JavaScript and expanding it are endless.

```
var speak = function(saywhat) {
  console.log(saywhat);
}

var Dog = function() {
  var name, breed;
}

var Cat = function() {
  var name, breed;
}

Dog.prototype.speak = speak;
Cat.prototype.speak = speak;

firstDog = new Dog;
firstDog.name = "Rover";
firstDog.breed = "Doberman";
firstDog.speak('woof');              //Returns the word woof.

firstCat = new Cat;
firstCat.name = "Sniggles";
firstCat.breed = "Manx";
firstCat.speak('meow');              //Returns the word meow.
```

2.4 Understanding call-and-apply Invocation

We've talked about almost all the ways to invoke functions except for one. And that is through two special methods available in JavaScript. Known as ***Call*** and ***Apply***.

http://www.w3schools.com/js/js_function_invocation.asp
In JavaScript, functions are objects. JavaScript functions have properties and methods.
call() and **apply()** are predefined JavaScript function methods. Both methods can be used to invoke a function, and both methods must have the owner object as first parameter.

call() example:

```
function myFunction(a, b) {
   return a * b;
}
myFunction.call(myObject, 10, 2);      // Will return 20.
```

apply() example:

```
function myFunction(a, b) {
   return a * b;
}
myArray = [10,2];
myFunction.apply(myObject, myArray);   // Will also return 20.
```

Both methods takes an owner object as the first argument. The only difference is that call() takes the function arguments separately, and apply() takes the function arguments in an array.
In JavaScript strict mode, the first argument becomes the value of ***this*** in the invoked function, even if the argument is not an object.
In "non-strict" mode, if the value of the first argument is null or undefined, it is replaced with the global object.

Note: With call() or apply() you can set the value of ***this***, and invoke a function as a new method of an existing object.
- When you use this technique, it's sometimes known as indirect invocation. That's because ***Call*** and ***Apply*** allow you to execute a

function in a slightly different way. And that gives you a little bit better control over the ***this*** argument.
- With ***Call*** and ***Apply***, you can define the value of the ***this*** argument. But in the traditional function declarations, the ***this*** parameter is bound to the global object.

```
var speak = function(what) {
  console.log(what);              //Returns the word moof.
  console.log(this);              //Returns the window object.
}

speak("moof");
```

The ***this*** argument returns the window object which is almost like saying, the entire browser. It's not really useful.

- You can control ***this*** and pass it along with some ***arguments*** as well, because ***Call*** and ***Apply*** take two parameters. The value of the ***this*** argument to be inside the function, as well as the ***arguments*** you want to pass to the function.
- Now the difference between the two is that ***Call*** passes a value and ***Apply*** lets you pass an array.

```
var speak = function() {
  console.log(this.love);         //Returns the word purr.
  console.log(this.normal);       //Returns the word meow.
}

var saySomething = {normal: "meow", love: "purr"}
speak.call(saySomething);
```

Taking saySomething object and passing it to the speak method as the ***this*** parameter.
So I was able to pass the object and I've got to output things and do things with this object by using the ***this*** parameter inside the function.

```
var speak = function(what) {
  console.log(what);              //Returns the word meow.
```

```
  console.log(this.love);              //Returns the word purr.
}

var saySomething = {normal: "meow", love: "purr"}
speak.call(saySomething, saySomething.normal);
```

I can also pass along a value when I'm setting to the ***this*** parameter. So the object that I am passing will be the value of ***this*** inside the function. What I can do is also pass along one of the values. And now it will get passed as the **what** parameter. So it gives you the ability to both pass something as the parameter, as well as control the value of ***this***. The ***this*** parameter, even though I'm using it to pass a value, the ***this*** parameter is still accessible.

As I mentioned, the difference between Call and Apply is that, if we switch the method to Apply, we will be able to pass along an array instead of an element. And then we still have access to saySomething object through the ***this*** parameter inside the function.

```
var speak = function(what) {
  console.log(what);                   //Returns the word meouff.
  console.log(this.love);              //Returns the word purr.
}

var saySomething = {normal: "meow", love: "purr"}
speak.apply(saySomething, ['meouff']);
```

So Call and Apply give you the power to bind objects of functions and include them as the ***this*** parameter. This makes functions even more flexible and is used in some important JavaScript techniques and patterns.

2.5 Using the Arguments Parameter

We've been passing a specific number of elements to functions, but oftentimes, you either don't know how many elements you'll be needing to pass to a function or you want your function to be able to accept any number of elements. For that, we can use a special object available to functions called the ***arguments*** parameter.
- the job of the arguments parameter is to hold a list of all the elements passed as arguments to the function.
- the arguments parameter is an array-like object, because it looks like an array and can do some of the things that we can do to arrays.
- We can call the arguments with a numerical index ***arguments[x]*** .
- We can also get the ***arguments.length*** , the property that gives us the amount of elements that were passed to the function.
- Because we have these two properties, we can loop through the arrays easily with a for loop.
- Because they're not arrays but an object that happens to have some list-like functionality, we can't use array methods like pop, push, shift, or others.

The for loop is going to go through all the elements in the argument array.
So as you can see, I can feed this function as many parameters as I want to.

```
var plus = function() {
  var sum =  0;
  for (var i = arguments.length - 1; i >= 0; i--) {
    sum += arguments[i];
  };
  return sum;
}

console.log(plus(2,2,2,3,2,3,4));
```

So here we're able to create a function that doesn't even require any arguments. It just accepts however many things you pass into it. With the arguments parameter, you can do this and other JavaScript patterns.

2.6 Returning Values

- Whenever you invoke a function, it's like generating an equation. And equation usually have results. So we call these statements, expressions. Because the job of the return statement, is to express the result of the operation we perform in our function.
- The return statement is kind of optional. If you don't include it in your code, the function will still return something. It'll just be the value undefined.
- The return statement is only available inside the function body. So you can't use it anywhere else.
- The return statement can be used to send something back to the caller. So you can assign it to a variable. And it'll place a value in the variable when the function returns.
- Return statements are usually the last statement in a function. But it doesn't have to be. You can put it anywhere in the function, but it will stop execution of the function.
- We can have as many return statements as you want. So if you want to, you can use if statements, to make the function stop. And return a value based on different conditions.
- The return statement can return anything or nothing at all. And that includes returning other functions, an object, or as I mentioned nothing at all. So for example, if you wanted to have a statement that evaluated the undefined, you can issue a return statement without an expression (return;) .
- You have to be careful with the return statement, is JavaScript auto semi-colon insertion. JavaScript has a weird feature that makes semi-colons on statements optional. It sort of tries to figure out where it should insert a semicolon. But since the return statement can be used without a value, it sees a return statement on its own as a complete statement.

```
Return
    a + b
```

Javascript will interpret the return statement as it's own statement. And then stop the execution of the function.

2.7 Social Media Navigation Example

```
<nav class="socialmediaicons"></nav>


var socialMedia = {
  facebook : 'http://facebook.com/viewsource',
  twitter: 'http://twitter.com/planetoftheweb',
  flickr: 'http://flickr.com/planetotheweb',
  youtube: 'http://youtube.com/planetoftheweb'
};

var socialList = function() {
  var  output = '<ul>',
    myList = document.querySelectorAll('.socialmediaicons');

  for (var key in arguments[0]) {
    output+= '<li><a href="' + socialMedia[key] + '">' +
      '<img src="images/' + key + '.png" alt="icon for '+key+'">' +
      '</a></li>';
  }
  output+= '</ul>';

  for (var i = myList.length - 1; i >= 0; i--) {
    myList[i].innerHTML = output;
  };
}(socialMedia);
```

3. Using Functions

3.1 Using Anonymous Closures

So far we've been using a JavaScript pattern that let's you create a variable, and then executes that as a function. So let's talk a little bit more about that, and then introduce you to a new pattern called an anonymous closure. Function definitions are really common and easy to understand.

1- Invoking the function by calling its name.

```
function kung() {
  console.log('foo');
}
kung();
```

2- Assign the function to a variable.

```
var iKnow = function kung() {
  console.log('foo');
}();
iKnow();
```

We also learned that all functions are considered objects in JavaScript. Since they are, you should be able to assign this function to a variable.
I can’t call it using ***kung();*** because the word kung doesn't exist in the global scope. So, now I have to call the function by the word ***iKnow***.

3- Letting the function execute itself by adding a pair of parenthesis to the function so that we don't have to call the function manually.

```
var iKnow = function kung() {
  console.log('foo');
}();
```

It also allows you to instantiate the function by typing in some variables between the parentheses so that we can use them inside the function.
The parenthesis changed our function declaration to a function expression.

4- Anonymous function.
We don’t need the function name because we're not using the kung function inside itself (like recursiveness). So we could get rid of its name and this will be an anonymous function now.

```
var iKnow = function() {
  console.log('foo');
}();
```

5- Anonymous closure.
Wrap the function part in parenthesis. That’s going to ask JavaScript to convert the function part into a value and also use the last parentheses as a way of passing some values to the function part if we wanted to.

```
(function() {
  console.log('foo');
})();
```

So this pattern is known as a self-executing function or an anonymous closure because it doesn't have a name and any variables you create inside it are only going to be accessible inside the function.
So, in other words, what we're doing here is we're closing the variables inside this function from the rest of the world, and that's why it's called a closure. This pattern is important when you work with certain JavaScript constructs, like modules.

3.2 Understanding Hoisting and Variable Scope

- Scope refers to when and where within your code a variable exists and retains a certain value (the life and death of a variable).

- In most programming languages, variables have a block scope. Variables created in a code block or inside curly braces exist only inside those curly braces {}. They cease to exist outside them.

- In function scope, variables live within functions. Any variables you create with the keyword ***var*** are local to the function that they were created in.

- Scope chain: if I have a function within a function. A variable you create in a parent function lives also in the child function as well. In simple terms, it's going to look for a variable up through all the parent functions.

- global variables: be careful with JavaScript variables, is that any variables that you create without the keyword ***VAR*** become ***global*** variables available to your whole application even if it's created inside a function. That's actually a very bad thing because a global variable can potentially cause problems when you have a lot of functions using similar variables. For example the ***dogName*** variable might collude with some other variable called ***dogName*** that you've declared at, in some other function. So, you never really want to expose your variables globally. You always want to use the ***VAR*** and make sure that you know where your scope of your variables is at any time.

- Variable definitions are hoisted. Remember, JavaScript is a scripted language, and, as such, during the browser-processing phase, JavaScript actually rearranges your variable declarations and moves them to the top of their functional scope. That means that variables can actually exist before you use them.

```
function myDog () {
   console.log(dogName + 'says woof');
   var dogName = 'Fido';
}

myDog();              //Returns   undefined says woof  .
```

We get ***undefined*** because JavaScript noticed that you declared a variable, and before it ran any of the code, it actually moved the declaration to the top of the function scope. So, it actually does something like this:

```
function myDog () {
   var dogName;
   console.log(dogName + 'says woof');
   dogName = 'Fido';
}
```

It actually creates a variable ***dogName***, and then assigns it to ***Fido*** that's why it says ***undefined***. The variable, it's actually sitting there, but it doesn't have a value yet.

And so, that can cause problems when you create more complex applications.

Say that you have a global variable ***dogName***, and named it Rover, and in the function, you try to declare the variable after you're calling it. So, what do you think should happen here?

```
var dogName = 'Rover';

function myDog () {
   console.log(dogName + 'says woof');
   var dogName = 'Fido';
}

myDog();              //Also returns   undefined says woof  .
```

You probably expect the console to log ***rover says woof***, but, we're going to get the same error. ***Undefined says woof***, because although the variable inside the function doesn't technically have a value until (after the log statement), the declaration has been moved to the top of the function (before the log statement). So, remember what's actually happening here is like you have a two separate statement.

This is how JavaScript is actually re-writing your code, and that can cause some additional problems because you may have something like this, not realize that later on, you have a variable that you created and expect it to have a different value.

The other interesting thing about this is that ***functions actually get hoisted as well to the top of the declaration keychain.***

So, for example, it doesn't actually matter where I put the function call. I can actually call it before the function exists (the function declaration).

So, that could cause some additional problems, and because of that, you must make sure that you declare all of your variables at the top of the function scope and also make sure you put all of your functions, if you can, also at the top of the function scope. The fact that you've got the function created and then you call it, is a good thing and a little bit better code and easier to read than what you would write otherwise.

3.3 Creating and Namespacing Modules

- Modules let you reuse code across different applications.

When you first start developing with JavaScript, you tend to focus on coding just for the current website. Whenever you begin a new site, you usually kind of look around your old code and grab a couple of different functions from here and there. And what modules allow you to do is to start reusing some of that code without having to copy and paste. It sort of creates libraries of things that you can use across different websites.

- The first thing you do whenever you create any module is create a namespace for it. Namespacing allows you to protect any variables that you have in your modules from any global scoped variables. This is important because in any module, you might be using some variables that are already used by other things in that application. So we already know that we can easily encapsulate our functions within parentheses, and that protects all the variables inside those functions from the global scope.

But what if we want access to those variables from within our application? To do that, we need to create a variable and assign it to our self-executing function. And that way we'll have access to things inside it.

```
var ray = (function() {

   var prvtVar;        //Those variables would not be accessible outside of this function. So they would be protected from the global scope.


   return {
   speak: function() {
    console.log('hello');
   }
   };

})();
```

- So now that we've namespaced the ***ray*** variable, we have a way to get into this function. But what about going the other way? What if we want to get something out of this function, or we want to execute something that's in this function, outside in the main application? Well, to do that, you're going to use a return statement that allows you to communicate back with the rest of the application. It can help you expose things that we want our application to know about. So instead of returning a statement, we can return an object, and through the object we can create variables and functions (methods).

Now I have a function that I can get to with the ***ray*** namespace and that I also can output something to the main application by creating this function. So I can call this function with that notation. We need to add a script tag that allows us to tell something to that function.

```
<script>
   ray.speak();        //Calling our method. Returns   hello  .
</script>
```

So, now we've created a module, and we're able to execute that module from within our main application, while still protecting all of our namespaces.

So this is a simple fully functional module. The key thing to note, though, is that we've added namespace which is going to protect the content within the module from any other global variables. Plus we used a return statement to execute functions we want our application to be able to execute.

3.4 Passing Arguments and Setting Module Defaults

In the last lesson, we created our module, and then used namespacing to protect any internal variables from outside scope. We also set up a return function, so we could talk to our methods. In this lesson, I'm going to show you how to send information to our functions using an object, and then how to set defaults, just in case the user happens to call a method without initializing it.
We can send things to our function easily through the arguments array. So right here, where we're calling the function, we can pass something to it. Now, I could just type in whatever I wanted the function to speak here. So I could say something like, `ray.speak( 'howdy' );` . But it's actually better to pass along things as an object. That way, we can pass multiple parameters into our function at the same time.

Create an internal object with a name called ***say*** and a value called ***howdy***.

```
<script>
   ray.speak({ say: 'howdy' });
</script>
```

I can easily capture the arguments by checking the ***arguments*** object that comes from our function.

```
var ray = (function() {

  return {
    speak: function() {
      console.log(arguments[0].say);
    }
  };
})();
```

The danger here is that the user executes a function, but doesn't pass anything you'll notice that it says that it can't say something undefined, because we're asking to read an argument that doesn't have anything in it. Since this empty call to our method is generating an error, we can fix this by using a little JavaScript trick called a **short-circuit evaluation**. If the user forgets to set it, the ***myArguments*** variable will be set to nothing.

NOTE
http://www.openjs.com/articles/syntax/short_circuit_operators.php
The && and || operators are called ***short-circuit operators***. They will return the value of the second operand based on the value of the first operand.

The && operator is useful for checking for null objects before accessing their attributes. For example...

```
var name = person && person.getName();
```

This code is the same as

```
if(person) {
   var name = person.getName();
}
```

The || operator is **used for setting default values**.

```
var name = persons_name || "John Doe";
```

The equalant code is

```
if(persons_name) {
   var name = persons_name;
} else {
   var name = "John Doe";
}
```

I can set up a default object for our function. And this is going to be available to any functions that I create for my project. So I'll set up a variable (object) called ***default***. And I'll set up a value there called say, as a default for what I want it to say.
And now it doesn't matter what I do, it's always going to output something correctly. And also I can always come up here and check my default values and modify them in one place.

```
var ray = (function() {
 var DEFAULTS = {
  say: 'hello'
 }

 return {
  speak: function() {
   var myArguments = arguments[0] || '';    //Set it to the arguments that
                                              get passed from my function, or set it to nothing.

   var statement = myArguments.say || DEFAULTS.say;
   console.log(statement);
  }
 };
})();
```

So now, you have a good module set up. You can call it with or without any parameters, and we have some defaults, just in case they're called without anything. It's a good idea to set up defaults for your modules at the top of your objects. They're going to be easier to find there, and can make your methods useful even if someone forgets to set them.

3.5 Chaining Module Method Calls

Our module works really well with a single function. In this lesson, I am going to show you a technique called ***chaining***. That you can use to make it easier to work with multiple methods. It's a really simple trick, that allows one function to call another. You've probably seen it in things like jQuery.

```
<script>
   ray.speak({ say: 'howdy' });
   ray.run({speed: 'fast'});          //Returns ***fast***, or ***normal*** by default if the
                                        user didn't provide an argument value.
</script>
```

So you may have seen some libraries that make this a little bit easier, and more concise. By letting you use dot notation, to chain one function call to another. Now this really couldn't be easier to set up. But the thing is, that you really have to understand why it's working. All we have to do is make sure each function returns the calling object. And Remember that's exactly what the ***this*** argument does in a function, return the object that contains it. Lets go back in our script. And what we'll do is, we'll issue a return statement inside each of these functions and have it return ***this***.

```
return this;
```

And now this means that I can execute these by putting them right next to each other.

```
<script>
   ray.speak({ say: 'howdy' }).run().speak({ say: 'run faster' }).run({speed:
'faster'});
</script>
```

Returns ***howdy***, ***normal***, ***run faster*** and ***faster***.

So why is this working? This is a little sort of weird. If you don't get what's going on. And the reason is, that the ***this*** parameter is going to return the instance of the object. So in other words, this is going to return the ***ray*** object, once again. Normally, return exits the current function. So, if I was out of the ***speak*** method. I wouldn't have the object that I needed to run the next method. But since I'm returning ***this***, the last thing that the ***speak*** function will do is return the object itself. And so therefore now, I have access back again to either my ***speak***, or my ***run*** function.
This technique is going to make your code just a little bit easier to call. And it makes it sort of fun to use. I've seen it used in libraries like D3 and jQuery. It doesn't take that much effort to set it up. Hopefully, after taking this course. You now understand why this is working. And you'll be able to utilize patterns like ***chaining***, and other ones. Because you understand them.

```
var ray = (function() {

  var DEFAULTS = {
    say: 'hello',
    speed: 'normal'
  }


  return {

    speak: function() {
      var myArguments = arguments[0] || '';
      var statement = myArguments.say || DEFAULTS.say;
      console.log(statement);
      return this;
    },

    run : function() {
      var myArguments = arguments[0] || '';
      var running = myArguments.speed || DEFAULTS.speed;
      console.log('running...'+ running);
      return this;
    }

  };
})();
```

4. Conclusion

I hope you learned something new and exciting about the functional nature of JavaScript and some of the cool patterns you can use to make your applications better.
There is a lot of really good books out there you can check out. So you may want to check out "***JavaScript: The Good Parts***" from Douglas Crockford. Everybody recommends his book and that's because it's a really awesome book, focusing on not a lot of stuff, but a lot of great stuff within JavaScript.
There's also this book that's actually being revised right now called "***Eloquent JavaScript***". If you're into a simpler explanation of JavaScript, this could be a really good book for you.
Also love to hear this podcast called ***JavaScript Jabber***. It has a lot of JavaScript focused content that you may enjoy.
Now I don't subscribe to a lot of newsletters but I really like this ***JavaScript weekly newsletter***. It gives you a lot of the top news happening in the JavaScript industry.
Now if you're into conferences, you may want to check out a ***O'Reilly Fluent***. It's a really good conference in San Francisco in March.
They always have a lot of really cool speakers.