Functions

And Scope!

What is a function?

- We've seen them before...what do you think they are for?

What is a function?

- A place to put code so that you have control over when it is executed.

- We can call our functions whenever we need them and use them over and over.

Function Syntax

```
function doSomething() {
    alert("something")
};
```

Where Should We Put Functions?

- Often functions go into .js files that are separate from our HTML. We include them in between the <head></head> tags of our HTML using <script> tags like this:

```
<script type="text/javascript" src="public/javascript/myFile.js"></script>
```

- We can also include javascript code directly in our HTML file like this:

```
<script type="text/javascript">
    function doSomething() {
        alert("something")
    };
</script>
```

How to Define a Function

```
function yay() {

    return "yay";

}
```

Function Return Value

- After a function is finished executing it will often leave us with a value. This is called *returning* a value. We use the *return* keyword to specify what we want to return.

```
<script type="text/javascript">
    function yay() {
        return "yay";    ⟵    The Return value is "yay"
    };
</script>
```

How to Call a Function

var celebration = yay();

console.log(celebration)

Exercise:

- Use the file "/materials/functions/exercises/nap.html"

- Make a function named *nap()* that returns the string "nap time!".

What if we want our function to use particular values while executing?

So far we have used static values for our functions to return. Programming gets much more interesting when we are dealing with variables that contain dynamic data.

Any ideas?

Arguments

- Arguments are cool because they make the same function return different things depending on which arguments we give it.

```
<script type="text/javascript">
    function yay(person) {
        return "yay " + person + "!";
    };
    console.log(yay("Scott");
    console.log(yay("Becky");
</script>
```

We define our function to accept an argument. The word *person* becomes the name of a variable that we can use in our function.

```
<script type="text/javascript">
    function yay(person) {
        return "yay " + person + "!";
    };
    console.log(yay("Scott");
    console.log(yay("Becky");
</script>
```

When we call our function, we can pass in an argument and our function will use it to give the *person* variable a value

Exercise:

- Using your *nap()* function add a *person* argument that you can use to make the function return strings like this:
  - "It's nap time for Scott!"
  - "It's nap time for Mary!"

Scope

where variables are available

Global Variables

- Assuming that the following variable is not inside of a function, it is a global variable and can be accessed from anywhere in your code:

```
var bikeColor = "green";
```

var Keyword & Variables

- If you define a variable inside of a function using the *var* keyword, it will be local, otherwise it will be global:

local variable

```
function bikesRock() {
    var bikeColor = "green";
}
bikesRock()
console.log(bikeColor);
```

global variable

```
function bikesRock() {
    bikeColor = "green";
}
bikesRock()
console.log(bikeColor);
```

Scope of Functions

```javascript
var globalVar = "globalVar";

function baseFunction() {
  var baseVar = "baseVar";



    function innerFunction() {
     var innerVar = "innerVar";
    }

}
```

baseFunction, defined in the global scope will have access to all global variables (globalVar) and its own local variables (baseVar).

baseFunction does not have access to any of the local variables inside of *innerFunction*

innerFunction, defined within *baseFunction* will have access to everything *baseFunction* has access to, plus its own local variables (innerVar).

Lexical Scope

- Javascript scope is determined by what variables are available in the definition of methods, not by what is available during execution.

- materials/functions/exercises/lexicalScope.html

Example:

- materials/functions/exercises/scope.html

Anonymous Functions

- Anonymous functions are dynamically declared at runtime and don't need to be given a name.

- Common usage:
  - Event listeners
  - Defining methods on an object
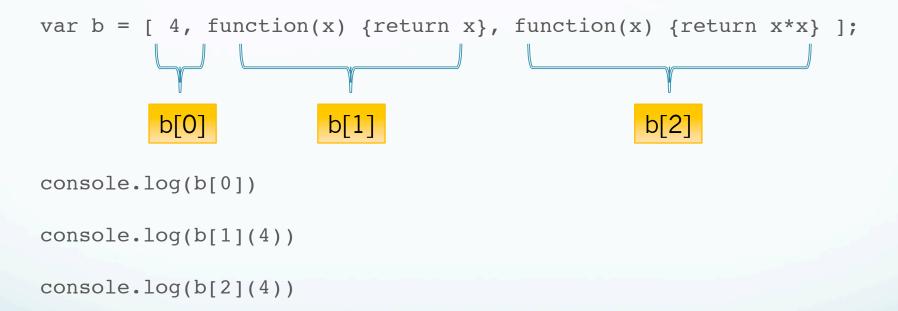
Something to Avoid...

- Don't do this:

```
var myFunction = new Function("name", "alert('hi!' + name);")
```

- Because:
  - Javascript has to parse the function body every time you call it.
  - You have to define the function body as a string.
  - The scope is global and can not inherit from the current scope chain.

Better Way to Make Anonymous Functions:

```javascript
var myFunction = function() {

                alert('yay!');

        }
```

Using Anonymous Functions

```
var b = [ 4, function(x) {return x}, function(x) {return x*x} ];
```

b[0]   b[1]   b[2]

```
console.log(b[0])

console.log(b[1](4))

console.log(b[2](4))
```

Exercise:

- Set a variable called *fruit* equal to an anonymous function that returns "apple"

Exercise:

- Set a variable called *fruit* equal to an array with three anonymous functions as elements. The anonymous functions should return "apple", "banana" and "orange"

Homework

- Read Chapter 7 (Events)

- Read Chapter 8 (Forms)

- AboutFunctions koan (first two tests, do the rest as extra credit if you wish)

- Do the exercises on page 120