Lesson 4

functions!

Learning Objectives

- Describe the purpose of functions in JavaScript
- Write and call a function
- Write and call a function that takes and uses arguments
- Write and call a function that return values
- Pass the results of a function call to another function
- Refactor code to make it cleaner and easier to read
- Understand variable scope, and global vs. local scope

Last week…

- Use conditionals to control program flow
- Differentiate between true, false, 'truth-y', and 'false-y'
- Use Boolean logic (!, &&, ||) to combine and manipulate conditionals

Fun with Functions

- A function is both a control flow structure AND a type of data -- the two things we've been talking about so far.

- Functions are my favorite part of JavaScript

- We are lucky that this incredibly powerful language structure, in its full unfettered glory, was included in this language with so many warts that was looked down on for so long

Demo: intro to functions

Intro to Functions

- A function is a reusable statement — or a group of reusable statements — that can be called anywhere in the program.

- No need to rewrite the same statements over and over.

- Functions enable software developers to break large, unwieldy programs into smaller, more manageable pieces.

DRY

- Functions enable a key tenet of engineering: Don't Repeat Yourself, or DRY.

- Our goal is to create programs with as little code as possible, while maintaining as much clarity as possible.

Functions are data themselves

- Functions are data, just like numbers, strings, and arrays. ‘function’ is a data type:

  typeof console.log === ‘function’

- In the same way that arrays are lists of items, functions are blocks of code.

- A function, being just another value that you can pass around, can also be passed into another function as an argument.

The official image of slides that say “functions are data”

Making our own functions

- Before we run a function, we must *define* it.
- In JavaScript, there are two very common ways to do this.
- We'll go over the simplest way first, which is to give it a name by assigning it to a variable.

```
var sayHi = function () {
  console.log("Hi, I'm Richard");
}
```

Calling our new function

- Running a function (also called "calling" it or "invoking" it) executes the code defined inside the function.

- **Defining** and **calling** a function are different. A function will not be called when it's defined.

- You call a function by using parentheses — **()** — after the function's name:

```
var sayHi = function() {
  console.log("Hi, I'm Richard");
}
sayHi();
// => Hi, I'm Richard
```

Anatomy of a function

1. The keyword **function**

2. Inside parentheses, an optional list of parameters — placeholder names for inputs that get passed into the function

3. Inside curly braces, the statements inside the function, which are the code that gets executed every time the function is called. (This is called the “function body”)

Anatomy of a function

- The parameter list below is empty because we're not giving the function any inputs, and

- the function body inside the curly braces consists only of the one statement **console.log("Hi, I'm Richard");**

- Then we give the function a name by assigning it to the variable **sayHi**

Functions as methods on objects

- JavaScript functions are often defined as methods on objects,

- This is what the **Math.floor** and **console.log** functions are.

```
var person = {
  name: 'Obama',
  speak: function () {
    console.log('Hello, World!')
  }
}

person.speak()
=> 'Hello, World!'
```

The official image of slides that talk about objects before I introduce them and at the same time I say “functions are data”

Alternative function declaration syntax

- Very common shorthand for defining functions and giving them names (assigning them to variables):

```
function sayHi() {
  console.log("Hi, I'm Richard");
}
```

- same as:

```
var sayHi = function () {
  console.log("Hi, I'm Richard");
}
```

Anatomy of the other kind of function declaration

1. The keyword **function**

2. **The new name of the function**

3. Inside parentheses, an optional list of parameters — placeholder names for inputs that get passed into the function

4. Inside curly braces, the statements inside the function, which are the code that gets executed every time the function is called. (This is called the “function body”)

Anatomy of the other kind of function declaration

- The parameter list below is empty because we're not giving the function any inputs, and

- the function body inside the curly braces consists only of the one statement **console.log("Hi, I'm Richard");**

- You don't have to assign it to a variable name because that's already taken care of with this syntax — the new name is **sayHi**

Codealong: intro to functions

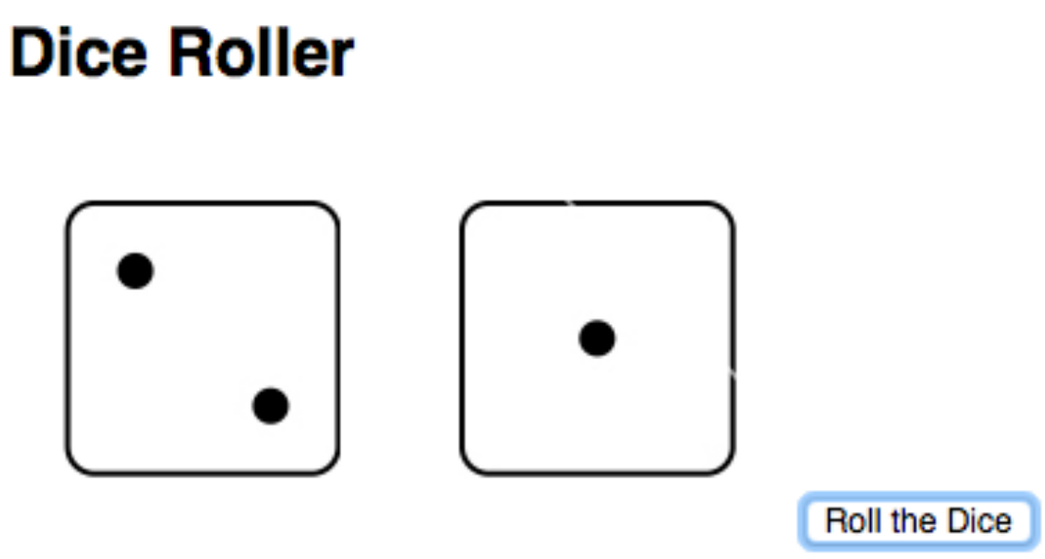
- Open a terminal.
- **cd ~/GA-JS**
- **mkdir lesson4**
- **cd lesson4**
- **touch functions.js**
- **subl functions.js**
- Go back and forth between the terminal and Sublime Text, editing the file and typing **node functions.js** in the terminal

Practice on your own: Intro to functions

- Write one function using function expression syntax and one using function definition syntax.
- Each of them can just console.log() something.
- Call them both.

Rolling Dice Lab

- For this lab, you'll be creating a page that displays a random update of two dice every time the user hits "Roll the Dice" button.
- Download the starter code from http://bit.ly/jsdev2-lesson4-starter
- Go into the js folder and open app.js



Demo: parameters

Why parameters?

- If a function did the same thing every time it was called, it wouldn't be a very useful codebase.

- We would have to write a new function for each new feature in order to enable additional behaviors in our program

```
// Bad idea...
function helloMark () {
  console.log('hello, Mark');
}

function helloObama () {
  console.log('hello, Obama')
}
```

What parameters do

- Parameters remedy this problem by allowing us to call the same function with different values

```
function sayHello (name) {
  console.log('Hello ' + name);
}

sayHello('Mark');
// => 'Hello Mark'

sayHello('Obama');
// => 'Hello Obama'
```

Parameter lists

- To write functions with more than one parameter, use a comma-separated list

```
function sum(x, y, z) {
  console.log(x + y + z)
}

sum(1, 2, 3);
// => 6
```

Parameters vs. arguments

- Parameters refers to variables defined in the function's declaration; arguments refers to the actual values passed into the function when the function is called. For example:

```
// the parameter here is myName
function sayMyName (myName) {
  console.log(myName);
}

// The argument here is "Bob"
sayMyName("Bob")
```

Codealong: parameters

Practice on your own: parameters

- Write a function that takes the name of a dish as an argument and prints out "And tonight for our special we have ..."

Demo: the return statement

The return statement

- Sometimes we don't want a function to print to the console or update the DOM or DO anything

- Rather, we just want it to give us back some data, when we run it.This requires a return statement.

```
function sum (x, y) {
  return x + y;
}

var z = sum(3, 4);
// => 7
```

The return statement

- When we return something, the function spits out the thing we are returning, which is called the "return value".

- We can then store the returned value in a variable, or do whatever else we want with it. We can even pass it to another function:

```
function sum (x, y) {
  return x + y
}

function double (z) {
  return z * 2
}

var num = sum(3, 4)
// => 7
var numDbl = double(num)
// => 14

// This can also be written:
var num = double(sum(3,4))
// => 14
```

Fun facts about return statements

- The return statement will completely stop a function's execution. Any statements following the return statement will not be called:

```
function speak (words) {
  return words;

  // The following statements will not run:
  var x = 1;
  var y = 2;
  console.log(x + y)
}
```

- By default, JavaScript functions will return the value **undefined**

Fun facts about return statements

- By default, JavaScript functions will return the value **undefined**

```
function what () {
  var nothingImportant = 8;
}
what();
// => undefined
```

Codealong:
return statements

Codealong: practice on your own

- Write a function that takes a word as its argument and returns a string with the word "lovely" in front of it. For example, if you pass "streetlamp" to the function, it will return "lovely streetlamp".

- Optional bonus: Write two functions that each perform some kind of basic arithmetic, and pass to the second function the result of calling the first function.

Interlude: refactoring

- Now that we know about parameters and return statements, can we get rid of some of the duplicated code in the rolling dice example?
- We will first do it by using parameters,
- Then we will see what we can do using the return statement

Demo: scope

Intro to Scope

- In any program, when we're executing one part of the code, other parts of the code may or may not be visible.

- By "visible", we mean that our part of the code has knowledge of, and access to, the variables that are defined in other parts of the code.

- When we declare variables inside functions, or when we use arguments inside functions, these things are NOT visible to the rest of the code, outside of the function.

- We say that those variables are in the function's "scope".

Global scope

- Before you write a line of JavaScript, you're in what we call the "global scope".
- Any variable you declare will be outside of any function, and is public, which we call "global".
- This means any other part of your code can reference this variable, meaning know that it exists, know what its value is, or change it.

```
// Outside of any function:
var name = 'Gerry';
// `name` is in global scope
```

Local scope

- Conversely, if a variable is declared inside a function, it is referred to as "local", and has a "local scope".

- A variable with local scope cannot be referenced outside of that function.

Local scope

```
var a = "this is the global scope";
function myFunction() {
  var b = "this variable is defined in the local scope";
}
myFunction();
console.log(b);
// => ReferenceError: b is not defined
```

- In this case, we get a reference error because the variable **b** is not accessible outside the scope of the function in which it is defined.

Access to the parent scope

- But a function can access variables of the parent scope. In other words, inside a function, you have access to all the variables defined in the global scope.

```
// Global Scope
var greeting = "Hello";

function sayHello(name) {
    return greeting + " " + name;
}

var fullGreeting = sayHello("JavaScript");
console.log(fullGreeting);
// => "Hello JavaScript";
```

Nested scope

- When a function is defined inside another function, it is possible to access variables defined in the parent from the child:

```
var a = 1;

function getScore() {
  var b = 2,
  c = 3;

  function add() {
  return a + b + c;
  }

  return add();
}

getScore();
// => 6
```

Codealong: scope