Grow with Google: Lesson 14

Functions

1. Intro to Functions

- Functions allow us to package reusable processes.
- Data is input, operations are performed, and the data is returned.

2. Reverse Function Example

- Functions prevent us from writing repetitive code.
- They allow us to reuse code, regardless of input.

3. Declaring Functions

- Functions can accept parameters, the data to be manipulated by the function.

```
// This function has no parameters.
function action () {
  // Do this.
}

// This function has one parameter.
function action (parameter) {
  // Do that.
}

// This function has two parameters.
function action (parameterA, parameterB) {
  // Do something.
}
```

- Functions usually expect some kind of explicit `return`.
  - When no return is included, `undefined` is returned instead.
- Functions must be invoked or called with arguments to run.

```
// declares the sayHello function
function sayHello() {
  var message = "Hello!"
  return message; // returns value instead of printing it
```

```
}

// function returns "Hello!" and console.log prints the return value
console.log(sayHello());
```

- Parameters and arguments can be differentiated based on where they appear in the code.
  - **Parameter** is a *variable* and appears in the function declaration.
  - **Argument** is a *value* and appears when the function is called.

4. Function Recap

- Functions allow the reuse of code.
- Parameters are variables that are used to store the data that's passed into a function for it to use.
- Arguments are the actual data that's passed into a function when it's invoked.

```
// x and y are parameters in this function declaration
function add(x, y) {
  // function body
  var sum = x + y;
  return sum; // return statement
}

// 1 and 2 are passed into the function as arguments
var sum = add(1, 2);
```

5. Quiz: Laugh it Off 1 (5-1)

```
function laugh() {
    var laughter = "hahahahahahahahahaha!"
    return laughter;
}

console.log(laugh());
```

6. Quiz: Laugh it Off 2 (5-2)

- The string and return had to be done outside of the loop.

```
function laugh(num) {
  var laughter = "";
  for (var i = 1; i <= num; i++) {
    laughter += "ha";
  }
    return laughter += "!";
}

console.log(laugh(10));
```

7. Return Values

- Returning a value and printing a value is not the same.
- When `console.log` is used, a value of `undefined` is returned.
  - A function is always going to return some value.
  - The `console.log` function accepts a parameter and prints the result to the console.
  - When no value is is specified, any function will return `undefined`.
- Return stops the execution of a function and returns a value.

```
function isPrime(integer) {
  for (var x =2; x < integer; x++) {
      if (integer % x === 0) {
          console.log(integer + " is divisible by " + x);
          return false;
      }
  }
  return true;
}
// isPrime(11); returns true.
// isPrime(49); logs "47 is divisible by 7" and returns false.
```

8. Using Return Values

- A function's return value can be stored in a variable or reused as a function argument.

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

```
function divideByTwo(num) {
  return num / 2;
}

var sum = add(5, 7);
var average = divideByTwo(sum);

// The stored value of average is 6.
```

9. Scope

- A point where a variable or function is still accessible.

10. Scope Example

- Global scope identifiers can be accessed from anywhere in the program.
- Function scope identifiers can only be accessed inside of a function.

```
var a = 1; // global variable
function x() {
  var b = 2; // local variable accessible to all functions below it
  function y() {
    var c = 3;
    function z() {
      var d = 4;
    }
    z();
  }
  y();
}
x();
```

11. Shadowing

- Shadowing occurs when global variables are overridden by subsequent functions.
  - To prevent overriding, a local variable of the same name can be declared as a new variable within the subsequent functions.

```
// Global
```

```
var x = 1; // x === 1

function addTwo() {
  x = x + 2; // x === 3
}
addTwo();
x = x + 1; // x === 4
console.log(x); // prints "4"

// Local
var x = 1; // x === 1

function addTwo() {
  var x = x + 2; // x === ?
}

addTwo();
x = x + 1; // x === 2
console.log(x); // prints "2"
```

12. Global Variables

- Global variables can conflict with other variables of the same name.
- It can get harder to track variables as programs get larger.

13. Scope Recap

- If an identifier is declared in global scope, it's available everywhere.
- If an identifier is declared in function scope, it's available in the function it was declared in (even in functions declared inside the function).
- When trying to access an identifier, the JavaScript Engine will first look in the current function.
  - If it doesn't find anything, it will continue to the next outer function to see if it can find the identifier there.
  - It will keep doing this until it reaches the global scope.
- Global identifiers are a bad idea. They can lead to bad variable names, conflicting variable names, and messy code.

14. Hoisting

- Before any JavaScript is executed, all function declarations are "hoisted" to the top of their

current scope.

- Hoisting happens with variables as well, but the assigned values do not.
- To avoid bugs, declare functions at the top of scripts, and variables at the top of functions.
- This ensures the way the code looks and behaves are consistent.

```
sayHi("Julia");

function sayHi(name) {
  console.log(greeting + " " + name);
  var greeting;
}
// prints "undefined Julia"
```

15. Hoisting Recap

- JavaScript hoists function declarations and variable declarations to the top of the current scope.
  - Variable assignments are not hoisted.
  - Declare functions and variables at the top of scripts, so the syntax and behavior are consistent with each other.

16. Quiz: Build a Triangle (5-3)

- I started a solution for this when I was tired and had to revisit after I had some rest.
- I spent some time figuring out wha the `makeLine` function actually was doing.
  - After testing the output with `console.log`, I learned it was drawing the base of the triangle, in full.
  - After sketching out triangles of varying lengths, a pattern emerged. The length of each line of the triangle was `length - (length-1)`. So when `length === 5`, the first line would equate to 1. The second line was `length - (length - 2)`, which equals 2, and so on.
  - As I was writing the code, however, I realized that I didn't need this equation, since each row of the triangle would be equal to the iterator anyway.
- My original solution had fewer lines of code but the lines weren't concatenated, so it didn't count as the correct size.
  - I was hoping to have the console log built into the `buildTriangle` function, but it wouldn't accept it as a valid solution.
  - The code fails without the return.

```
```

```
function makeLine(length) {
    var line = "";
    for (var j = 1; j <= length; j++) {
        line += "* ";
    }
    return line + "\n";
}

function buildTriangle(length) {
    var triangle = "";
    for (var i = 1; i <= length; i++) {
        triangle += makeLine(i);
    }
    return triangle;
}

console.log(buildTriangle(10));
```

17. Function Expressions

- Function expressions are functions stored in a variable.
  - These expressions do not need a name since it would be redundant given the var name.
  - This is called an anonymous function.
  - Accessing the variable with the stored function will return the function.
- Unlike function expressions, function declarations are not hoisted.
  - Variable declarations are hoisted, but variable assignments are not.

18. Patterns with Function Expressions

- A function that is passed into another function is called a callback.
- Function expressions can be given a name.
- Functions can be passed inline.
- Anonymous inline functions are often used with function callbacks that aren't going to be reused.
  - Save lines of code with inline functions.

```
// Callback Version
// Function expression that assigns the function displayFavorite
// to the variable favoriteMovie
```

```
var favoriteMovie = function displayFavorite(movieName) {
  console.log("My favorite movie is " + movieName);
};

// Function declaration that has two parameters: a function for displaying
// a message, along with a name of a movie
function movies(messageFunction, name) {
  messageFunction(name);
}

// Call the movies function, pass in the favoriteMovie function and name of
movie
movies(favoriteMovie, "Finding Nemo");

// Inline Function Expression Version
// Function declaration that takes in two arguments: a function for displaying
// a message, along with a name of a movie
function movies(messageFunction, name) {
  messageFunction(name);
}

// Call the movies function, pass in the function and name of movie
movies(function displayFavorite(movieName) {
  console.log("My favorite movie is " + movieName);
}, "Finding Nemo");
```

19. Function Expressions

- **Function Expression:** When a function is assigned to a variable. The function can be named, or anonymous. Use the variable name to call a function defined in a function expression.

```
// anonymous function expression
var doSomething = function(y) {
  return y + 1;
};

// named function expression
var doSomething = function addOne(y) {
```

```
  return y + 1;
};

// for either of the definitions above, call the function like this:
doSomething(5);
```

- You can even pass a function into another function inline. This pattern is commonly used in JavaScript, and can be helpful streamlining your code.

20. Quiz: Laugh (5-4)

- This was pretty straightforward and required no planning at all.

```
var laugh = function(num) {
    var laughter = "";
    for (var i = 1; i <= num; i++) {
        laughter += "ha";
    }
    return laughter + "!";
}

console.log(laugh(10)); // hahahahahahahahahaha!
```

21. Quiz: Cry (5-5)

- Also pretty straightforward, although functions without parameters always feels weird.

```
var cry = function weep() {
    var tears = "boohoo!";
    return tears;
}

console.log(cry()); // boohoo!
```

22. Quiz: Inline (5-6)

- Got a little confused by the syntax at first.
- I also didn't realize I had to use "happy" as the string initially.
- I still think this seems ridiculous to save a few characters.

```
```

```
function emotions(myString, myFunc) {
    console.log("I am " + myString + ", " + myFunc(2));
}

emotions("happy", function laugh(num) {
        var laughter = "";
        for (var i = 1; i <= num; i++) {
            laughter += "ha";
        }
        return laughter + "!";
    }
);
```

23. Lesson 5 Summary

- Next up: arrays.