JAVASCRIPT FUNCTIONS

MICRO-REVIEW FROM PROGRAMMING LESSON

- What is a program?
- What are variables for?
- What is the purpose of a conditional?

ONE COOL THING TO TACK ONTO YESTERDAY'S CONDITIONALS: TERNARY OPERATORS!

- And if/else statement as we know it works perfectly. But there are some times where you can use a shorthand form of if else statements: the ***ternary operator.***
- The ternary operator looks like this:

```
conditional ? Code if true : code if false
```

- Helpful if you have a small if/else statement. Never needed, but sometimes cool.

WHAT IS A FUNCTION?

- In it's most basic form, ***a function is a reusable statement.***
- This avoids the need to rewrite the same statement over and over
- Functions enable the software developer to segment large, unwieldy applications into smaller, more manageable pieces.
- critical component of programming, functions address a key tenet of engineering: Don't Repeat Yourself, or DRY.

DRY

- We should always be striving to be creating DRY code.
- Our goal is to create programs with as little code as possible, while maintaining complete clarity.
- If you do notice that you are doing some piece of logic over and over again, you should be thinking “how do I make that into a function?”

HOW TO DECLARE A FUNCTION: FUNCTION EXPRESSION

- One way to create a function is with something we call a *function declaration*
- *A function declaration is where we declare a function to a variable name.*
- Like this:

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

HOW TO DECLARE A FUNCTION: FUNCTION EXPRESSION

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

- This looks like a normal variable, has the same “var” declaration, and the “=“ assignment
- The only difference is that we are using the keyword “function” to announce that instead of declaring the variable as a value, we are declaring it as a function.

HOW TO DECLARE A FUNCTION: FUNCTION EXPRESSION

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

- After the keyword “function” there are some clappers with a parameter inside (more on that in a few minutes)
- After the clappers, we put a curly bracket that contains the code that we want to run on the next lines.
- ***That code is indented inside by one tab or two spaces.***

THE ONE PROBLEM WITH FUNCTION EXPRESSIONS

- Function expression suffer from the same problem that variable declarations have.
- You can only use the function AFTER you declare it.

```
speak('hello, world!');

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

// results in an error
```

THE ONE PROBLEM WITH FUNCTION EXPRESSIONS

- The only way to use it is if you call the function after it is declared
- Not a huge deal, but something to be careful with.

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

speak('hello, world!’);
// returns ‘hello, world!’
```

HOW TO DECLARE A FUNCTION: FUNCTION DECLARATIONS

- Another way of declaring a function is with function declarations:

```
function speak(words) {
  console.log(words);
}
```

- In a function express you begin with the keyword function

HOW TO DECLARE A FUNCTION: FUNCTION DECLARATIONS

- Take a good hard look, it takes some getting used to:

```
function speak(words) {
  console.log(words);
}
```

- Note that with a function declaration, there is no equal sign assigning the value. You just begin with the keyword “function” and javascript expects the next value to be the name for it

HOW TO DECLARE A FUNCTION: FUNCTION DECLARATIONS

- Take a good hard look, it takes some getting used to:

```
function speak(words) {
  console.log(words);
}
```

- A function declaration always has the following:
- A name
- An optional list of parameters (i.e., the names of arguments to be "passed" into the function, or information the function will use); this is defined by the parenthesis before the opening curly brace
- Statements inside the function (the code executed every time the function is called)

FUNCTION DECLARATIONS: HOISTING

- One of the wonderful things about a function declaration is that javascript will read any function declarations into memory first. No matter what.
- This means that you don't necessarily need to declare the function before using it!

```
speak('hello, world!');
//it works!


function speak(words) {
  console.log(words)
}
```

FUNCTION DECLARATIONS: HOISTING

- This characteristic of function declarations is called “hoisting”
- This makes things a bit easier to work with. For this reason, function declarations are widely preferred over function expressions.

```
speak('hello, world!’);
//it works!


function speak(words) {
  console.log(words)
}
```

CALLING A FUNCTION – "INVOKING"

- Calling, or invoking, a function executes the code defined inside this function.
- But defining and calling a function is different. A function will not be called when it's defined.
- You call a function by using parenthesis after the function's name () (or as I call, "clappers" 👏👏)

```
function hello() {
  console.log("hello there!");
}

hello();
```

LET'S REFLECT

- What is the point of declaring functions?
- Parts of declaring a function: "function", clappers, parameters function name, {} , code block.
  - In a function expression, in what order are these parts used?
  - In a function declaration, in what order are these parts used?
- What is hoisting?
- What does it mean to invoke a function? How do you do it?

FUNCTION EFFECTIVENESS

- Now a function like that hello() function isn't very useful on it's own.
- Sure now we can do this:

```
function hello() {
  console.log("hello there!");
}

hello();
hello();
hello();
```

- Why is that a bad idea?

FUNCTION EFFECTIVENESS

- Now a function like that hello() function isn't very useful on it's own.
- Sure now we can do this:

```
function hello() {
  console.log("hello there!");
}

hello();
hello();
hello();
```

- Why is that a bad idea?

FUNCTION EFFECTIVENESS

- It's just not practical! We could get the same thing done with less code like this:

```
console.log("hello there!");
console.log("hello there!");
```

FUNCTION EFFECTIVENESS

- It's just not practical! We could get the same thing done with less code like this:

```
console.log("hello there!”);
console.log("hello there!”);
console.log("hello there!”);
```

- If we were to make functions for everything, we would have a messy, inefficient ugly bit of code.
- ***Functions have one purpose: take an input, perform logic you'd want to do multiple times, output the result.***

FUNCTION PARAMETERS

- So how do we allow different inputs into our functions?
- This is remedied by “parameters”. Some also call them “arguments” interchangeably, and honestly no one really cares. But teeeechnically speaking:
  - Parameters are used to define a function;
    - As in “I created a function with these parameters”
  - Arguments are used to invoke a function.
    - As in “I passed these arguments into the function”
- Cool that’s totally not that important, and to be honest, 99% of developers don’t know the difference. Welcome to the 1% you knowledge-wizards you. Anyways, moving on…

USING FUNCTION PARAMETERS

- As we were saying “parameters” are how we allow functions to take in a variety of inputs to perform a small program on them.

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

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

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

USING FUNCTION PARAMETERS

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

- Here the parameter is called “name”.
- The parameter is declared inside of the clappers after the keyword ‘function’ and after the name of the function is declared.
- This parameter can be thought of as a variable that is special to that function. This special variable is created during the function declaration! No “var” or anything required.

USING FUNCTION PARAMETERS

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

sayHello('Trevor')
=> 'Hello Trevor'
```

- Then when you invoke the function, whatever you put in the clappers will replace where the parameter is found inside of the function.

USING FUNCTION PARAMETERS

- Functions can have any many parameters as you want. You just separate them with commas.
- When you invoke the function, the order of the parameter is respected by the order of the arguments you put into it.

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


sum(1, 2, 3);
=> 6
// x = 1, y = 2, z = 3.  In the
same order as the parameters!
```

USING FUNCTION PARAMETERS

- Parameters can take in any data type, strings, numbers, booleans, even arrays and objects.

```
function printAnimals(animalArray) {
  for(var i=0; i < animalArray.length; i++) {
    console.log('I like ' + animalArray[i]);
  }
}

printAnimals(['chicken', 'penguins', 'velociraptor']);
```

THE RETURN STATEMENT

- So far we've been doing console.logs for us to see values in the console. This is and will always be a very useful development tool. Keep using it.
- That said, it doesn't actually MEAN anything. It's just printing information for us as we develop.
- Sometimes we don't want to just print to the console, we might instead want to update a variable, use the output from a function, or event call another function.
- This requires a 'return' statement

THE RETURN STATEMENT

- When we return something, it ends the function's execution and "spits out" what we are returning. We can then store this returned value in another variable...

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

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

- Using console.log(x + y) here would not have allowed us to assign the result to a variable name

THE RETURN STATEMENT

- Note that 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)
}
```

THE RETURN STATEMENT

- By default, JavaScript functions will return an undefined value. To test this, use Node to define and run a function without a return value. A return value "overwrites" this default value.

LET'S REFLECT

- When should we use a function?
- What are function parameters and what is there purpose?
- What is the difference between a parameter and an argument?
- What is 'console.log' for?
- What is the return statement for?

CODEALONG

- The value of a return statement gets more evident as we get into more realistic code examples.

- Open up the main.js file included in this repo

LAB

Let's practice

SCOPE

- We had discussed earlier how we can think of parameter as special variables that belong to a function. There is a bit more to this.
- ***parameters can not be accessed outside of the function***

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

console.log(speak('hedgehog'); //works!!
console.log(words);  // does not work
```

LOCAL SCOPE

- Additionally, any variables that are declared inside of a function can not be accessed from outside the function. We call this “scope”.
- ***A variable is always “scoped” inside of the function that it is declared.*** Meaning that it can only be accessed inside of that function, or any functions declared inside of that function (yes that’s a thing).
- A variable scoped inside of a function is said to have ‘local scope’

SCOPE

```
var myNum = 4;


function multiplier(num) {
  var multiplier = 8;
  return multiplier * num;
}


console.log(myNum);  // => 4
console.log(multiplier); // undefined
```

GLOBAL SCOPE

- Before you write a line of JavaScript, you're in what we call the Global Scope. When a variable is declared outside a function, it is public—referred to as GLOBAL—and has a global scope. Any script or function on the page can then reference this variable.
- For example, when you declare a variable right away, it's defined globally:

```
var name = 'Gerry';
```

- This name variable is not inside a function, and is therefore in the ‘global scope’. That means that any function on the page can use the variable.

GLOBAL SCOPE

- Global scope can be confusing when you run into namespace clashes. ***You don't want to use global scoping for all your variables.*** This can lead to a lot of confusion.
- Most of the time it is a good idea to have most of your program compartmentalized into functions, and each function should have its own scope.

LOCAL AND GLOBAL SCOPE USAGE

- Let's run the code below in our codealong js file:

```
var a = "this is the global scope";
function myFunction() {
  var b = "this variable is defined in the local scope”;
  console.log(a);
}
myFunction();
console.log(b);
```

- In this case, the console log will send a reference error because the variable b is not accessible outside the scope of the function in which it is defined.

LOCAL AND GLOBAL SCOPE USAGE

- As we said earlier, A function can access variables of the parent scope. In other words, a function defined in the global scope can access all variables defined in the global scope.

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

// This function is defined in the global scope
function sayHello(name) {
    return a + " " + name;
}

sayHello("JavaScript");
=> "Hello JavaScript";
```

NESTED FUNCTION SCOPE

- When a function is defined inside another function, it is possible

```
var a = 1;


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

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

  return add();
}

getScore();
=> 6
```

LET'S REFLECT ON EVERYTHING FROM THE LESSON

- When should we use a function?
- What does it mean to invoke a function and how is it done?
- What is hoisting? What kind of functions are hoisted?
- What are function parameters and what is there purpose?
- What is the difference between a parameter and an argument?
- What is 'console.log' for?
- What is the return statement for?
- What is local scope?
- What is global scope?
- Is it good or bad to put things in global scope?