FUNCTIONS

WHAT ARE FUNCTIONS?

- A function is a piece of code that you can **invoke/call** from an another section of code.
- JavaScript has many built in functions that we've used
  - Math.random(), Math.round(), etc.
  - Console.log(), document.getElementById()
- When using functions written by others, you need to know what the function does, but not how it is implemented
- This allows people to share their work or work together in a convenient way, and each person can work on their own set of functions

BENEFITS OF FUNCTIONS

- Better program organization
  - Some function we only use once
  - But by giving sections of code meaningful names, our code is more readable.
- Easier to test
  - Can isolate small sections of code to test.
- Reusing your code
  - Solving problems once and use the solution again!

MORE BENEFITS

- Allows us to use the event driven model – next week
- It does take a little more effort to write programs using functions
- Overall, you will save time by breaking down your program into smaller functions.
- The extra design time is well worth the effort

CREATING FUNCTIONS

- In JavaScript there are two ways we can create functions, but all functions have the same basic anatomy
- The first way looks like this:

```
function myFirstFunc (param1, param2){
    //your code goes here
    return //something - like a value;
}
```

ANATOMY OF A FUNCTION

The word “function” starts all JavaScript functions

```
function function_name (arg1,arg2, …)
{
      //Do something interesting here
       return <return value>;
}
```

ANATOMY OF A FUNCTION

A unique name that others parts of the code can refer to the function by

```
function function_name (arg1,arg2, …)
{
      //Do something interesting here
       return <return value>;
}
```

ANATOMY OF A FUNCTION

Optional list of Input parameters

```
function function_name (arg1,arg2, …)
{
      //Do something interesting here
       return <return value>;
}
```

ANATOMY OF A FUNCTION

```
function function_name (arg1,arg2, …)
{
      //Do something interesting here
       return <return value>;
}
```

function body

EXAMPLE

- Suppose we wanted to make a function to add two numbers together
- What two numbers?
  - Any two. Those numbers would be the **parameters** of our function
- What would be a good name for our function?
  - add? Probably!
- What is the output of our function?
  - The sum of the two parameters
- Let’s write it!

AN ADD FUNCTION

```
function
```

AN ADD FUNCTION

```
function add
```

AN ADD FUNCTION

```
function add(num1, num2)
```

AN ADD FUNCTION

```
function add(num1, num2){

}
```

AN ADD FUNCTION

```
function add(num1, num2){
     return num1 + num2;
}
```

- **Congrats!** You just wrote your first function.
- Let’s test it in the console

CREATING FUNCTIONS

- The second way we can create a function looks like this:

```
let myFirstFunc = function(param1, param2){
      //your code goes here
      return //something - like a value;
}
```

- This form emphasizes the fact that functions are primitive types in JavaScript (like Number, String and Boolean)

EXAMPLE

- The same addition function we wrote before could also look like this:

```
let add = function(num1, num2){
     return num1 + num2;
}
```

- NOTE: The body should be enclosed in curly brackets, even if your function is only one line!

NAMING FUNCTIONS

- You can’t just call your functions anything you want:
- There are definite rules and requirements you MUST follow
- When you create function, its name:
  - **SHOULD** not be an existing keyword
  - **CANNOT** have spaces
  - **CANNOT** use \ or $, (,),+,-,{,},[,],’,”,,,.,?,:,; etc..

NAMING FUNCTIONS

- There are conventions or soft requirements
- These are guidelines that you should follow
  - Do not use non-english characters
    - The rational is that not all editors support those characters or have the font to display them
    - Using these characters makes it hard for others to read them
  - You should give your function a descriptive name
    - sumOfCubes(x,y) ✓
    - isValidNumber(num) ✓
    - myFunction() ✗
  - Avoid all special characters except _ (underscore)
  - Use the camelCase naming convention

YOUR TURN

- Write a function that accepts a string representing a name as a parameter. The function should return the string “Hello NAME, have a good day”

- Write a function that accepts a temperature in Fahrenheit as a parameter and returns the temperature in Celsius
- C = (F – 32) * 5 / 9

DOING MATH IN JS

- There are a ton of built in functions we can use in JavaScript

- Some great math functions that we can use
- **`Math.pow(5,2)`** `returns` 5^2
- **`Math.round(2.66)`** `returns 3`
- **`Math.round(2.33)`** `returns 2`
- **`Math.random()`** `returns a random number between 0.0 and up to but not including 1.0`

DOING MATH IN JS

- More Math!

- **`Math.ceil(5.3) returns 6`**
- **`Math.floor(10.7) returns 10`**
- **`Math.min(2,4,6) returns 2`**
- **`Math.max(2,4,6) returns 6`**

GENERATING SPECIFIC RANDOM NUMBERS

- We can use Math.random() to generate random numbers that fall between specific digits
- For example, if we want to generate a random **integer** between 0 and 100

```
let random = Math.random() * 101;
random = Math.floor(random);
```

GENERATING SPECIFIC RANDOM NUMBERS

- For example, if we want to generate a random **integer** between 1 and 100

```
let random2 = Math.random() * 100;
random2 = Math.floor(random2);
random2 = random2 + 1;
```

MANIPULATING STRINGS IN JS

- String Functions
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

- `let str = “hello my name is Inigo Montoya”;`
- `str.length;`
  - will return the length of the string (how many char, including spaces)
- `str.indexOf("a");`
  - will return the position of the first "a" in the str
- `str.lastIndexOf("a")`
  - returns the position of the last "a"

MANIPULATING STRINGS IN JS

- `let str = “hello my name is Inigo Montoya”;`
- `str.substring(6,16);`
  - returns “my name is”
- `str.substring(17);`
  - returns “Inigo Montoya”
- `let newStr = str.replace("Inigo", "Inconceivable")`
  - newStr will now contain the string “hello my name is Inconceivable Montoya”

MANIPULATING STRINGS IN JS

- `let str = "Inigo Montoya";`
- We can convert a string to uppercase and lowercase

- `let bigStr = str.toUpperCase();`
  - bigStr will be "INIGO MONTOYA“

- `let lilStr = str.toLowerCase();`
  - lilStr will be "inigo montoya“

- `str.charAt(0);`
  - will return I

MANIPULATING STRINGS IN JS

- `let str = "Inigo Montoya";`
- We can convert a string to uppercase and lowercase

- `let bigStr = str.toUpperCase();`
  - bigStr will be "INIGO MONTOYA"

- `let lilStr = str.toLowerCase();`
  - lilStr will be "inigo montoya"

- `str.charAt(0);`
  - will return I

```
let name = "";
name += bigStr.charAt(0);
name += bigStr.charAt(6);
name += lilStr.substring(7,11);
```

Now what is stored in name?

EXERCISES

- Write a function that accepts two numbers as parameters and returns the sum of their cubes, i.e., $x^3 + y^3$
  - Ex. sumOfCubes(2,3) should return 35

- Write a function that accepts a string as a parameter and returns the string with proper capitalization (i.e., the first letter capitalized and the rest of the word lowercased)
  - Ex. properCase("AbCdEf"); should return "Abcdef"

WHEN DOES A FUNCTION RUN?

- Remember that
  - JavaScript runs when a webpage is parsed
  - JavaScript runs in response to an event

- But when does a function run?
  - **Functions do not run when they are defined**
  - **Functions run when they are called**

USING THE FUNCTION

- When we want to use the function, we must **call** or **invoke** the function
- We invoke a function by calling its name and the brackets the with parameters in it
- Examples:

```
let num = sumOfCubes(2,3);
console.log(properCase("aaBBcC");
```

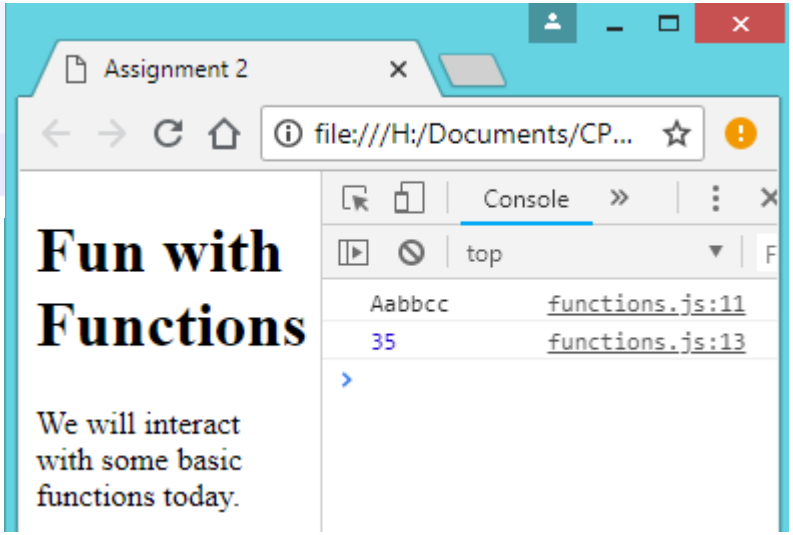
- After the function has finished executing, we return back to where the function was called, and continue on from there
- Let's see it in action: functionExample.html

THE RETURN VALUE

- When we call a function, it is **evaluated to a value**
- Sometimes we don't care what it evaluates to
- When we do care about what a function evaluates to, we need to pass that information back to where it was called, so we can store it and use it again

```
11  console.log(properCase("aaBBcC"));
12  let sum = sumOfCubes(2,3);
13  console.log(sum);
```

These function calls evaluate to a value, and then that value is printed to the console



PATH OF EXECUTION

```
1  function sumOfCubes(num1, num2){
2      return Math.pow(num1, 3) + Math.pow(num2, 3);
3  }
4
5  function properCase(str){
6      let result = str.charAt(0).toUpperCase();
7      result += str.substring(1).toLowerCase();
8      return result;
9  }
10
11 console.log(properCase("aaBBcC"));
12 let sum = sumOfCubes(2,3);
13 console.log(sum);
```

TERMINOLOGY

```
1  function sumOfCubes(num1, num2){
2      return Math.pow(num1, 3) + Math.pow(num2, 3);
3  }
4
5  function properCase(str){
6      let result = str.charAt(0).toUpperCase();
7      result += str.substring(1).toLowerCase();
8      return result;
9  }
10
11 console.log(properCase("aaBBcC"));
12 let sum = sumOfCubes(2,3);
13 console.log(sum);
```

FUNCTIONS

MAIN PROGRAM

WHAT ARE PARAMETERS?

- When writing a function parameters are like place holders
  - They are similar to variables in that they contain values, and give those values a name
  - We do not know what are inside parameters until our code runs, which gives our code flexibility

- Parameters are useful because can often solve a problem **generically**.
- Then, when we want a specific solution, we can call our function and specify the parameters to solve for

EXAMPLE OF PARAMETERIZATION

- Consider calculating the perimeter of a circle (circumference)
- We can find the circumference for ANY circle using the equation, yes?
- How?

EXAMPLE OF PARAMETERIZATION

- Consider calculating the perimeter of a circle (circumference)
- We can find the circumference for ANY circle using the equation, yes?
- How?

- **`2*Pi*r`**

- We know how to calculate this circumference, even if we don't know what the radius r actually is because **r is a parameter**

EXAMPLE OF PARAMETERIZATION

- Then, when we do finally know the radius of the circle we want to calculate the circumference of
  - Say we used a tape measure and measured the radius
- We just plug the radius r into our **function** 2*pi*r
- We can be fairly confident that we have the correct circumference if we measured the radius correctly
- We can also be confident that our function will work for any valid value for **r**

CIRCUMFERENCE

```
function circumference(radius){
    return 2 * Math.PI * radius;
}
```

- Then if we want to actually calculate using radius = 5 we just call
  - `circumference(5);`
- Whereas if we want to calculate for a circle with radius 15
  - `circumference(15);`

BUT WHAT ARE THE PARAMETERS?

- For **primitive types** the parameters (and return statement) can be treated as passing by value
  - That is to say, we make a **copy** of the primitive for use in the function
- Passing in **functions**, **objects** and arrays have slightly different behavior we will be discuss later
- A **parameter** is an input to a function
- This allows our function to me more flexible, and compute values or perform actions based on the parameter.

THE POWER OF PARAMETERS

- Parameters allow our functions to be more re-usable
- Or at least more useful, overall even within out program
- We don’t have to write the same code over and over again
- We can write one function and just change the parameters every time we want to use it

SCOPE

```
function <function_name> (<arg1>,<arg2>, …)
{
    let i;
    //Do something interesting here
    return <return value>;
}
```

Any variable declared/created inside of a function is a local variable

VARIABLE SCOPE - LOCAL

- When we define a variable inside a function with the keyword **let** it is only visible inside the function
- These variables are called **local variables**
- You cannot access these variables from outside of the function
- Local variables are created on the fly, every time the function is called
- And then these local variables are destroyed once the function has finished executing

VARIABLE SCOPE - GLOBAL

- When we define a variable outside of a function, it is a **global variable**
- Every function can see this variable, use it's value, and change it's value
- Global variables exist throughout your entire JavaScript
- Your different functions have the ability to modify global variables

IDENTIFY THE SCOPE

```
1  function sumOfCubes(num1, num2){
2      return Math.pow(num1, 3) + Math.pow(num2, 3);
3  }
4
5  function properCase(str){
6      let result = str.charAt(0).toUpperCase();
7      result += str.substring(1).toLowerCase();
8      return result;
9  }
10
11 console.log(properCase("aaBBcC"));
12 let sum = sumOfCubes(2,3);
13 console.log(sum);
```

IMPORTANT NOTE

- Functions created using the
  let myFunc = function()
  way must be declared BEFORE you invoke them!

- JavaScript runs top to bottom, so trying to call this type of function before it exists will cause ERRORS

EXAMPLE

```
circumference(100);
let circumference= function(radius){
   return 2 * Math.PI * radius;
}
```

Bad

```
let circumference = function (radius){
   return 2 * Math.PI * radius;
}
circumference(100);
```

Good

HOWEVER

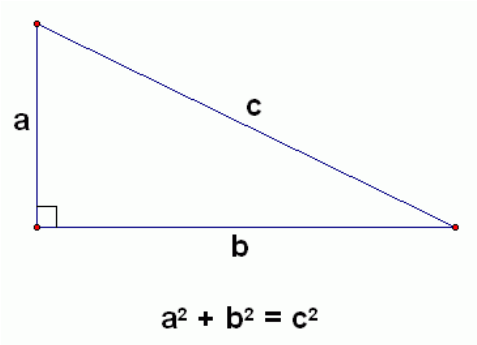
- Functions created using the function declaration way are not part of the regular top-to-bottom flow of control
- they are conceptually moved to the top of their scope and can be used by all the code in that scope

- This is sometimes useful because it gives us the freedom to order code in a way that seems meaningful, without worrying about having to define all functions above their first use.

EXERCISE

```
function hypotenuse(a,b){
   return Math.sqrt(Math.pow(a,2) + Math.pow(b,2));
};
let a = 3;
let b = 4;
console.log(hypotenuse(a,b));
```

- What is written to the console?



FUNCTIONS AND THE DEBUGGER

- debugger.html
- If we examine the execution path in the debugger
  - We see that during a function call, the code jumps to part of the source code.
  - After the function finish executing, it returns back to the where function was originally called.
- Chrome and Firefox both have good debuggers

EXERCISE

```
function doubleOrTriple( x ) {
    if(typeof(x) !== "number"){
        return NaN;
    }
    if(50 <= x){
        return x*2;
    } else {
        return x*3;
    }
}
```

What do the following two expressions return?

```
doubleOrTriple(40);
doubleOrTriple(doubleOrTriple(40));
```

EXAMPLE

```
function testFunction(b){
   b = 6;
 console.log(b);
};
let outerVar = 5;
testFunction(outerVar);
console.log(outerVar);
```

- What is the output after the code is finished executing?

EXAMPLE

```
function testFunction(b){
   b = 6;
 console.log(b);
};
let outerVar = 5;
testFunction(outerVar);
console.log(outerVar);
```

Prints

6

5

- Note that the **value of outerVar did not change**, even though **b is changed** inside the function
- This demonstrates the idea of **pass-by-value**
- We can safely assume that this behavior will happen whenever we are passing primitive types as our parameters

MISSING PARAMETERS

- JavaScript does not enforce the number of parameters you invoke a function with
- You can define a function with 2 parameters and give it 0 parameters when you call it
- If this sounds like it might be a **problem**, then you are **right**!

```
function runMe(in){
    console.log(in);
}
runMe(10);
runMe();
```

EXERCISE

```
let x = 9, Y = 18, Z = 27;
function a (x) {
      let y = 2;
      return x + y + z;
}
function b (y) {
      let z = 19;
      return x + y + z;
}
function c (z) {
      let x = 3;
      return x + y + z;
}
```

What is the value of each of the following function calls?

- `a(10);`
- `b(10);`
- `c(10);`