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

The word “function” starts all JavaScript functions

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

The word “function” starts all JavaScript functions

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

Optional list of Input parameters

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

Anatomy of a Function

The word “function” starts all JavaScript functions

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

Optional list of Input parameters

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

function body

Example

- Remember the lab exercise where we had to calculate the sum of cubes?
- $sum = x^3 + y^3$
- We can write function to calculate this for us

```
function sumOfCubes(x, y){
   var sum = Math.pow(x,3)+Math.pow(y,3);
   return sum;
}
```

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)

Anatomy of a Function

The word “function” starts all JavaScript functions

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

Optional list of Input parameters

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

function body

Example

- The same sum of cubes function we wrote before could also look like this:

```
let sumOfCubes = function(x, y){
   let sum = Math.pow(x,3) + Math.pow(y,3);
   return sum;
}
```

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

Naming Your 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 Your 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

Exercise

- Write a function that prints out the numbers 1 to 100 in the console

```
function printTo100(){
  for(var i = 1; i <= 100; i++)
    console.log(i);
}
```

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
- We can find the perimeter for ANY circle using the equation 2*Pi*r
- We know how to calculate this perimeter, even if we don't know what the radius actually is
- When we do finally know the radius of the circle we want to calculate the perimeter for
  - Say we used a tape measure and measured the radius
- Then we just plug the radius r into our formula, because ***r is a parameter***
- We can be fairly confident that we have the correct perimeter if we measured the radius correctly

Perimeter of a Circle

```
function circlePerimeter(radius){
   console.log((2 * Math.PI * radius));
}
```

- Then if we want to calculate the perimeter of a circle with radius = 5 we just call
  - `circlePerimeter(5);`
- Whereas if we want to calculate for a circle with radius 15
  - `circlePerimeter(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

Exercise

- Now modify your print to 100 function so that it accepts a number as a parameter, and instead of printing 1 to 100, the function prints from 1 to the number given as a parameter

```
function printTo(max){
   for(var i = 1; i <= max; i++)
      console.log(i);
}
```

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

```javascript
let drawingSurface = document.getElementById("myCanvas");
let ctx = drawingSurface.getContext("2D");


function drawSomething(ctx,x,y,N,r1,r2){
    let xOrigin = 50, yOrigin = 25;
    if(isNaN(x) || isNaN(y) || isNaN(r1) || isNaN(r2) || isNaN(N)){
        console.log("Error in one of the parameters.");
        return;
    }
    ctx.save();
    ctx.translate(xOrigin,yOrigin);
    ctx.beginPath();
    for(let i = 0; i <= N; i = i + 1){
        ctx.rotate(Math.PI/N);
        ctx.lineTo(r1,0);
        ctx.rotate(Math.PI/N);
        ctx.lineTo(r2,0);
    }
    ctx.stroke();
    ctx.restore();
    return;
}
```

In this JavaScript code, which of the variables are global, which are local and which are parameters?

Anatomy of a Function

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

Return statements are optional
When you do use them, they are usually the last line of your function

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

The Return Value

- When we call a function, it is evaluated to a value
- Sometimes we don't care what it evaluates to
  - We using these functions to carry out work such as drawing something
- However sometimes it's very useful

- Remember back to the perimeter calculation, 2*Pi*r
- Instead of printing the calculation to the console, it would be sensible to return the result of the perimeter calculations
- If we return the value, then we can use it again!

Perimeter of a Circle

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

- Then in order to store the calculated perimeter so we can use it again
  - `let p1 = circlePerimeter(5);`
  - `let p2 = circlePerimeter(15);`

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
- Example:

```
circlePerimeter(10);
```

- After the function has finished executing, we return back to where the function was called, and continue on from there

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:

Bad

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

Good

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

However

- Functions created using the function declaration way do not follow this rule
- function declarations 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.

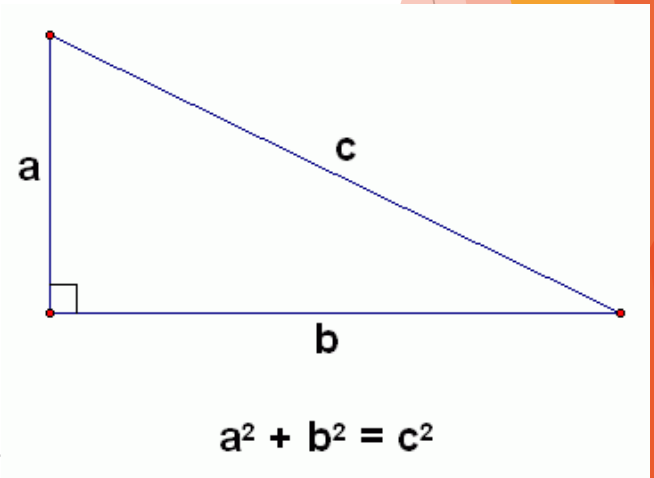
Fine

```
circlePerimeter(100);
function circlePerimeter(radius)
{
    return 2 * Math.PI * radius;
}
```

Exercise

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

- What is written to the console?



Exercise

```
function doubleOrTriple( x )
{
    "use strict";
    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));
```

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

Example

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

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

Example

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

- Note that the value of the **outerVar** does not change, even though **b** is changed inside the function
- We call this **pass by value**
- We can safely assume this when we deal with **primitive types**

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();
```

Validating inputs

- We need to validate inputs
- Which means we have to check if the input contains what we expect!
- Because JavaScript is weakly typed, we can get a string when we expect a number, and many other unexpected inputs

Validating inputs

- Useful functions for validating input:
- Our good old friend isNaN()
- typeof()
  - Tells us what type of value the variable contains
  - “number”, “string”, “boolean”, “undefined”, ”object”

- the following example will print “number”

```
let input = 15;
console.log(typeof(input));
```

Function with Validation

```
function AreaOfCircle(r){
    if(typeof(r) === "number" ){
        return Math.PI*Math.pow(r,2);
   } else {
        return NaN;
   }
};
```

- **Always** validate if you are accepting raw input from user.
- No exceptions, ever!
- Log an error message to the console if you do not get the input you expect

```
let x = 9;
let y = 18;
let 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;
}
```

Exercise

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

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

Exercises

- Write a function that accepts two parameters, m and n, and returns the sum from m to n
  - ex. calling sum(5, 10) would return 45

- Write a function that accepts two parameters, number and factor. Your function should return how many times factor evenly divides into number. If number is not evenly divisible by factor, you should return -1
  - ex. calling numOfFactors(20,10) should return 2
  - ex. calling numOfFactors(15,4) should return -1