3. Functions and conditional statements

3.1 What are functions? Why do we need them?

- **What are functions/methods?**
    - **Functions/methods:**
        - Both are basically the same and mean a set of instructions or procedures we design and group together to perform a specific task. You can describe functions as words (specifically verbs) are for the English or any human language. Writing a function means we are defining some code/procedures that we can use later in our program whenever we want to
        - Example: If we say "RUN" in English, it is basically a set of instructions grouped together. The word run can be broken down into smaller tasks:
            - Stand on your right leg
            - Then, lift your left leg
            - Move your left leg in front of your right leg
            - Stand on your left leg
            - Lift your right leg
            - Move your right leg in front of your left leg
            - Now, do the above steps repeatedly and faster (walking needs to be defined too)
    - **Difference between functions and methods**:
        - A function lives on its own.
        - A method is a function that belong to an object. A method is built in JavaScript and is associated with an object property. Do not worry, we will cover objects and methods in a separate class later.
- **Why do we need functions?**
    - Like stated above, we write functions (set of instructions) to perform a specific task and use that instruction whenever we want that task to be done in future. Therefore, functions are very important to avoid writing the same set of instructions again and again, every time we want the same kind of task to be done.
    - To allow us use one function over and over, we give our function a name. The next time we want to perform the same kind of task, we just call the name of the function

and use it. Please note that functions need inputs to successfully perform their task. The inputs we give to our functions are called "Argument".

- We can see reusing of specific instructions in human language too. For instance, in English, if one says "paint", the instruction is not specific enough because,
    - Some might paint a wall, some might paint on a canvas, some might paint with color red, some might paint with yellow
    - But if one says, "paint the wall blue",
        - The word "paint" is the function/instruction/ the instruction one needs to perform
        - Wall and blue are the arguments, inputs to the function Paint
        - We can use the instruction "paint" over and over by changing "blue" and "wall" values

3.2 How do we declare/define a function in JS?

- **What does declaring/defining a function mean?**
    - **Definition**: Declaring a function is simply writing all the code/instruction for your function. At that point the function just sits there doing nothing. When you declare a function, you are basically telling the JS compiler about the name of your function, what the function returns, and the parameters named in the function.
- **How do we declare a function?**
    - Declaring a function uses the keyword "function", just like we used the keyword "var" to declare a variable. Then we need to provide the name of the function. The name can be whatever you choose it to be.
    - We cannot have two functions with the same name. Then we tell if the function needs an argument to perform its task.
    - Example, a function that does addition, needs the numbers to be added. If it does, we provide "parameters" within brackets to the function. Parameters are just variables/names that we use when we declare a function, just to say our function needs arguments to perform its task. If a function doesn't provide parameters, we just leave the brackets empty. Then we put all the instruction we want the function to do with inside curly braces.

- **Function declaration vs function expression**: There are two ways of defining/declaring a function, we can either use a function declaration or a function expression.
    - **Function declaration: Syntax for defining function with declaration**

```
function myFirstFunction (number1, number 2) {
  return number1 + number2;
}
```

        - **Note**: Declared functions are not executed immediately. They are "saved for later use", and will be executed later, when they are invoked (called upon).
        - **Call/invoking a function in JavaScript using the call() method**
            - The code inside a function is not executed when the function is defined. The code inside any JavaScript function will execute only when "something" invokes it.
            - It is common to use the term "call a function", "invoke a function", "start a function, "start a function, all to mean execute the function.
            - See below how we can call/invoke the above example function:
                - myFunctionName(11, 10); // returns 21 because of 11 + 10
    - **Function expressions**: **Syntax for defining function with declaration**

```
const x = function (number3, number4){
return number3 + number4
  };
```

        - After a function expression has been stored in a variable, the variable can be used as a function. Functions stored in variables are anonymous or do not need function names. They are always invoked (called) using the variable name
            - You can call/invoke function x above like this:
                - x (9, 11) // this will return 20, because of 9 + 11

3.3 Functions with arguments

- **Parameter vs argument of a function**:
  - **Function parameters**: Some functions take inputs to successfully perform their task. The input we give to our function is called "parameter". Function parameters are the names/variables given when we define/create the function. They are just values we supply to the function so that the function can do something utilizing those values.

```
Example 1:
function myFunction (a) {
console.log (a);
}
```

- In the above example, “a” is the input/parameter that we gave for the function called, myFunction
- JavaScript function definitions do not specify data types for parameters.
- **Note**: Functions can be declared wihtout parametes as well

```
function mySecondFunction () {
  console.log (“Hello world”);// prints Hello world
  }
```

  - **Function arguments**: Arguments are the actual value of the parameter passed to and received by the function. Arguments are values we supply when we call/use/invoke a function.
    - Example: When we invoke the myFunction function above, we will need to pass an argument like this

```
myFunction (“Hello world”)// prints Hello world
```

- **Note 1**: JavaScript doesn’t check the number of parameters you define and the number of arguments you pass. This means, JavaScript functions can be called with any number of arguments, regardless of the number of parameters we named when defining the function originally.

- For example, let's say you create a function with ONLY two parameters. You can call this function and pass in 10 arguments. JavaScript will not care. It will happily invoke the function, create and assign variables for all parameters and execute the function. Not a single error thrown.

```
    function myFunc(param1, param2, param3) {

  return [param1, param2, param3]
    }
myFunc(1, 2, 3, 4, 5) // outputs [1, 2, 3] with no error
```

- **Note 2**: What if you pass fewer arguments than there are parameters? Those arguments you omit will be set to undefined. Example:

```
function myFunc(param1, param2, param3) {
  return [param1, param2, param3]
  }
myFunc('one', 'two') // outputs [ 'one', 'two', undefined]
```

- **Note 3**: The points under note 1 and note 2 above is simple, when you invoke a function, make sure to pass all required arguments. Remember that JavaScript will not warn you when you miss some arguments or pass more arguments than is necessary.

▪ **How do function parameter and function argument work in JavaScript?**

- **When you pass arguments to a function**:
  - JavaScript will create new variables using names of the parameters
  - JavaScript then give these variables initial values using the argument values you passed when you called the function.
  - These variables will be local to the function. They will exist only inside it (you will not be able to access any of these variables from the outside)

- These variables will exist only during the function call. Once the function call is finished, these variables will be lost.

```
Example:  function adder (x, y) {
  console.log (x + y)
}
adder (5, 4) // prints 9, the sum of 5 and 4
```

  - When we called/invoked adder function, JS created/declared var x and var y
  - JavaScript then assigned an initial value of 5 to x and an initial value of 4 to y

3.4 Functions that return a value

- In real world, we don't usually want the results to be displayed in a console window. So, what happens is, our function returns the calculated value to who ever wanted the result. Some functions don't return a significant value, but others do. It's important to understand what their values are, how to use them in your code, and how to make functions return useful values.
- **Return values**: These are values that a function returns when it has completed.
- **Syntax for functions that return value**: A return statement is used in a function body using the key word "return". See syntax below:

```
function myFunction (myParameter){
  return value;
  }
```

- **Logging a function's value in the console vs returning a function's value**
  - **console.log()**: It is a method that prints any kind of variables defined before in it or to just print any message that needs to be displayed to the user. It doesn't have anything to do with relaying information from one part of your program to another. console.log evaluates its argument and prints it, continuing the execution of the function it's called from.

- **Return**: A return value is used where the function is an intermediate step in a calculation of some kind. After a function calculates the value, it can return the result so the result can be stored in a variable; and you can use this variable in the next stage of the calculation. See the code below to understand how we can use a return value of a function to calculate something else:

```
 function myAdditionCalculator (a, b) {
  var c;
  c = a + b;
  return c;
}
Now, let us use the returned sum to do something
else
     var numberOne;
      var numberTwo;
      var  total;
        numberOne = 4;
     numberTwo = 5;
total = myAdditionCalculator (numberOne,
numberTwo);
```

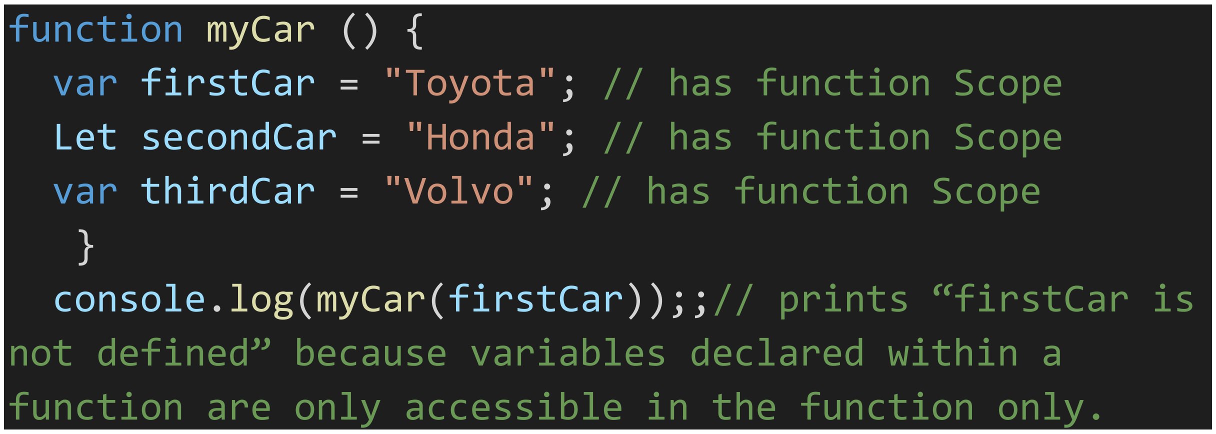
- **Note**: The return statement stops the execution of a function and returns a value from that function. Let's see that in example:

```
function printDog() {
  console.log(“dog'”);
  return;
  console.log(“cat”);
}
```

  - If you then invoke printDog(), you'll see the output dog in the console, but you won't see the output cat, because return ends execution of printDog() before it can get to console.log('cat');

3.5 Variable scoping

- **Scope**: Scope in JavaScript is a rule that manages the availability of variables. In short, scope determines which part of your code can see/use or access your variables.
- **Lexical scoping**:
  - Just like many modern programming languages, JavaScript also follows the lexical scoping of variables. Lexical scoping is the way by which we set scope of a variable so that it may only be called (referenced) from within the block of code in which it is defined. This means that functions are executed using variables that were named when a function is declared, not using variables named when the function is called/invoked.
  - In short, any function, no matter the place where being executed, can access the variables of its lexical scope
- **The different forms of variable scoping in JavaScript**
  - **Function scope**:
    - Each JavaScript function creates a new scope. Variables declared within a JavaScript function become LOCAL to the function, meaning variables within a function are only recognized in that function. That is why function parameters are always local to that function.
    - Variables declared with var, let and const are quite similar when declared inside a function.



```
function myCar () {
  var firstCar = "Toyota"; // has function Scope
  Let secondCar = "Honda"; // has function Scope
  var thirdCar = "Volvo"; // has function Scope
   }
  console.log(myCar(firstCar));;// prints “firstCar is
not defined” because variables declared within a
function are only accessible in the function only.
```

  - **Global scope**: The global scope is the outermost scope. Variables declared globally (outside any function), are recognized by everyone and accessible from any inner (local) scope.

- **Automatically global variables**: If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable in JavaScript. See the following example code for explanation:

  // code can use firstName here or anywhere globally because JavaScript will declare a global variable firstName, even if the value is assigned inside a function

```
function myFirstName() {
  firstName = "Alem";
  }
  myFunction();
```

▪ **Block scope**: JavaScript introduced block level scope after 2015 (ES6). In ES6, const and let keywords allow JavaScript developers to declare variables in the block { } scope, which means those variables exist only within the corresponding block. In block scope, variables defined in a block of code are only accessible within that same block.

- **Variables declared with var**: variables declared with var inside a block { } can NOT have block scope, meaning, they can be accessed from outside the block.
- **Variables declared with let & const:** These variables create block scope variable. This is how most other programing languages scope variables). It is recommended to use let instead of var. let prevents mistakes of trying to redeclare an already declared variable
- **Variables declared with const:** const is like let but the value remains constant always and you cannot reassign the value. const variables need to be declared using an initializer, or it will generate an error

```
if (true) {
  const message = “Hello”;
  var otherMessage = “Hi”
  console.log(message); // 'Hello' // message is
accessible
```

```
  }
  console.log(message); // throws ReferenceError because
message is not available outside the block
  console.log(otherMessage); // no error and logs “hi”
because variables declared with var inside a block are
available outside the block
```

3.6 Arrow functions

- Arrow functions were introduced in ES6. An arrow function is an alternative to a traditional function expression but is limited and can't be used in all situations. Arrow functions allow us to write shorter function syntax.
    - **Syntax**: const myFunction = (a, b) => a * b;
- **Arrow function vs traditional function**:

```
Traditional:
const hello = function() {
  return "Hello World!";
}
Arrow:
const hello = () => {
  return "Hello World!";
}
```

Arrow function with ONLY one statement and ONLY one return value: In this case, you can remove the brackets as well as the return keyword:

```
const hello = () => "Hello World!";
```

- **Arrow Function with Parameters**: Look at the function below that has one parameter called val

```
const hello = (val) => "Hello " + val;
```

- In fact, if you have only one parameter, you can skip the parentheses as well:

```
const hello = val => "Hello " + val;
```

3.7 Understanding statements: conditional statements

- **What are statements?** In a programming language, statements are instructions we write to interact/instruct with computer/browser so that the computer/browser produces the result we want it to produce. The entire goal of learning a programing language is to be able to communicate with computers and use their computational power to solve problems of the world. One of the most challenging things for new programmers is not actually grasping the techniques of writing the code. Unfortunately, many newcomers focus only on this part. What is most important is understanding the "sequential instruction" nature of programing languages.
    - Asking a computer to do something complex is like instructing a three-year-old child on the other side of the wall to carefully follow your instructions and draw a floor map of a big house. Before you can figure out how to communicate with the child and successfully instruct him to draw a floor map, you should first figure out how to make him draw a single line. Then may be some curved line. You should also have feedback mechanism to see if the boy is drawing what you are instructing him to draw. Eventually, once you figure out his level of communication skill, you can adjust your way of communication to his level to speed up the communication between the two of you.
    - Let's think of a different scenario. Let's say the person on the other side of the wall is a professional architect. How would things be different? In this scenario, you don't have to oversimplify your way of communication as you did for the child.
    - Now, let's change the child or the architect to a computer. How much do you have to simplify your way of communication for the computer to understand you? Understanding this is SUPPER important. Plus, how do you talk to something non-human? To help you understand this concept of "sequential instruction" and help you adjust to the communication level of computers as of 2020, think of Chu'lo the Robot, as the one you are talking to. You can find Chu'lo here: https://www.evangadi.com/courses/Chulo.jpg
    - Chu'lo is a robot that lives on the other side of your computer. He is only three years old and doesn't understand English. He understands functions as you understand words in English. He is super obedient and does exactly what you ask him to do. NOTHING MORE NOTHING LESS!!

    - So, it is very important that you write your instructions in a sequential manner, that include each step to achieve the goal, whenever you write instructions in a form of JavaScript code. When writing your code, just make sure they are clear enough that even a child can understand them.

- **Steps of making JavaScript decisions**: In any programming language, the code needs to make decisions and carry out actions accordingly depending on different inputs. We need decisions because the code we write can usually take more than one direction depending on different situations. To determine which direction our code should take, we need to have decision-making mechanism. These are the common steps we take while we want to make any kind of decision.
    - **Evaluation:** This is when you analyze values in your script to determine whether they match expected results or not.
        - Example 1: Hey chu'lo, create a folder called "Abebe" if it doesn't already exist
        - Example 2: If student scores 50 or more, it means score is a pass
        - Example 3: If student scores less than 50, it means score is a fail
        - score = 82; // should output pass
    - **Decision**: Using the results of the evaluation, you will decide which path to take.
        - Example 1: Chu'lo checks if there is a folder named "Abebe" and makes decision to create the folder or not
        - Example 2 & 3: The decision is to determine if student got a pass or fail score
    - **Action**: This is what you do after the decision is made.
        - Example 1: Creating the folder for example
        - Example 2 & 3: Passing or failing the student
    - **Loops**: On occasions where you must do the evaluation and decision steps repeatedly, you use loops. Loops are used in JavaScript to perform repeated tasks based on a condition in a more manageable and organized manner.
        - Example: Hey Chu'lo, create 100 files in the current directory

Conditional Statements: Conditional statements are used to perform different actions based on different conditions. Conditional statements are used to specify if a block of code should be executed if a specified condition is met/true.

- **There are three parts of conditional statements in JavaScript**:
    - **The Evaluation**: In this process, the comparing of values is done to make a decision. Example:

```
var a = 82;
(a <= 55)
```

-
    - **The Decision**: Once the evaluation is done, we go to the value of our decision, This is always a True or False value. Example:

```
var a = 32;
(a <= 55)
Decision: True
```

-
    - **The Statement**: This step will determine what to do when conditions are fulfilled or not fulfilled. Example:

```
console.log("RAISE YOUR HAND!");
console.log("You failed");
```

- **In JavaScript we have 4 conditional statements**: we will discuss all of them in the following sections of the note
    - if statements
    - else statements
    - else if statements
    - switch statements

3.8 if statements

- **Definition**: An "if" statement is a programming conditional statement that we use to specify a block
- **Syntax**: We start by using the key word "if", put the conditions within a bracket. Note that if is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error.

```
    Syntax 1:
if (condition) {
  // write the code you want to be executed if
the condition is true
}
   Example:
var pass = 50;
var score = 17;
if (score >= pass) {
console.log("Hey you passed");
}
if (score <= pass) {
console.log("You failed");
}
```

- **Syntax 2 (alternative syntax)**: If the statement is only a single line, you can leave out the containing curly braces
  - Example:

```
if (score <= pass) console.log("You failed");
```

3.9 else statements (“else” and “else … if” statements)

- **The “else” statement**
  - **Definition**: We use the “else” statement to specify a block of code to be executed when the condition under the “if” statement is false.

```
Syntax: if (condition) {
  // write the code you want to be executed if the
condition is true
} else {
  // write the code you want to be executed if
condition is false
}
```

```
•   Example:
var score = 82;
if (score >= pass) {
console.log("Hey you passed");
}else{
console.log("You failed");
}
```

- **The "else … if" statement**
  - **Definition**: There will be times where you want to test multiple conditions. That is where the else if block comes in. When the "if" statement (the first statement) is false, the computer will move onto the "else if" statement and executes it.

```
   Syntax 1: if (condition 1 is true) {
  // code is executed
} else if (condition 2 is true) {
 // code is executed
} else {
  // code is executed
}

  Example: Let's instruct Chu'lo the computer
If (checkIfFolderExists('Abebe')) {
console.log("Folder already exists");
}else{
createFolder('Abebe');
}
```

  - **Syntax 2 (conditional or ternary operator/ alternative for if … else)**: The conditional (ternary) operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and finally the expression to execute if the condition is falsy. This operator is frequently used as alternative to an if...else statement.

condition ? expressionIfTrue: expressionIfTrue

```
Example:   var age = 43;
     var theDrink= " ";
    theDrink = (age >= 18) ? "Alcohol is allowed" :
"Only juice or water";
console.log(theDrink); // "Alcohol is allowed"
prints because 43 >= 18
```

3.10 Switch statements

- **Definition**: instead of using this long if else statement, you might choose to go with an easier to read switch statement. The switch statement starts with a variable called the switch value. Each case in "switch statement" indicates the possible value and if there is a match when comparing the input value with values of a case, the code associated with the match will run.

```
   Syntax:
switch(expression) {
 case x:
// the code you want to be executed if expression
matches value of the case
    break;
  case y:
// the code you want to be executed if expression
matches value of the case
break;
 default:
// the code you want to be executed if expression
matches value of the case
}
```

- **Notes on how “switch statements” work**
  - The expression in a “switch statement” is evaluated only once with each case
  - In a “switch statement”, the comparison is between the expression and the values under each case
  - If there is a match between the expression and the value of any of the cases, the code block associated will be the one to be executed
  - If multiple cases match a case value, the first case is selected.
  - If there is no match while comparing the expression and the value of each case, the default code will be executed

```
Example for “switch statements”:
var greetings; var timeOfDay; timeOfDay =
"afternoon";
switch (timeOfDay) {
case "morning":
greetings = "Good morning";
break;
case "afternoon":
greetings = "Good afternoon";
break;
case "evening":
greetings = "Good evening";
break;
default:
greetings = "Hi there";
break;
}
console.log(greetings);  //
```

- This prints in the console "Good afternoon" after comparing the expression value in the switch statement (value of timeOfDay) with each case. We see that there is a match when the case is “afternoon”. Therefore, the code in that associated with the matching case has executed.

- **What does the keyword “break” do in “switch statements”?**
  - When JavaScript reaches a break keyword, it breaks out of the switch block. This will stop the execution inside the switch block. **Note**: It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.
  - **What does the keyword “default” do in “switch statements”**?
    - Default clause in switch statement works as the same as else clause in if else block. The “default” keyword specifies the code to run if there is no match between the condition and the expression.
    - The default statement doesn't have to come at the end. It may appear anywhere in the body of the switch statement.
    - If no default label is found, the program continues to the statement(s) after the switch. If default is not the last case in the switch block, remember to end the default case with a break.

```
   Example:
let x = 4;
switch (x) {
case 0:
console.log("Off");
break;
case 1:
console.log("On");
break;
}
console.log("hiiiiii"); // Because there is no
default case, this code will run and print "hiiiiii"
```

- **When do we choose “switch” statements over “if ...else” statements?**
  - Unlike “if... else” statements, we use “switch statements” ONLY when we know the precise value of condition/ input/expression we are going to compare. For example, in the below example, we can’t use score > 70 as a condition in “switch

statements” because we don’t know the precise value of score > 70. It will log the default value "Score is 70 or lower"

Example 1 (incorrect way to evaluate imprecise value with switch statement):

```
var score = 80;
switch(score){
    case score > 70:
    console.log("Score is higher than 70");
break;
    default:
     console.log("Score is 70 or lower");
}
```

Example 2 (if you want to evaluate an imprecise value using the switch statement, you need to create a workaround by evaluating a true expression)

```
var score = 70;
switch(true){
    case score > 70:
console.log("Score is higher than 70");
break;
    default:
console.log("Score is 70 or lower");
}
```