2. Introduction to programming basics (part II)

2.1 What is data structure and why do we need to structure data?

- **Data structure**: It is a particular way of storing, organizing and processing data/information in a computer so that it can be retrieved and used most productively.
    - The concept of data types is important in programming. That is why learning programing language starts by understanding what data is and how it is structured for effective use. To be able to operate on variables, it is important to know something about the data type.
    - Data structures are the building blocks of algorithms and computer programs. It is a key topic when it comes to Software Engineering interview questions. Hence as developers, we must have good knowledge about data structures.
- **Why do we need to structure data in the first place?**
    - Like we have said earlier, data is the only thing computers understand. We need to give them a lot of data for them to be able to understand us and our world better. If we just store data in the computer without any organizational structure, it quickly becomes very hard to use. It becomes very difficult to RETRIEVE and use it later when we need to use it. Hence, we need to organize/structure the data we store in our computer.
        - Example: Without data types, a computer cannot safely solve this:

```
var x = 14 + "Hello";
```

    - Data Structures are necessary for designing efficient algorithms. It provides reusability and abstraction. Using appropriate data structure, can help programmers save a good amount of time while performing operations such as storage, retrieval or processing of data.

2.2 Types of data structures

- **Data structure types**: There are two types of data structure available for the programming purpose, primitive data structure and non-primitive data structure.
    - Today, many programming languages include an extensive collection of built-in data structures to organize code and information. For example, in Python: lists and dictionaries, and in JavaScript: arrays and objects are common coding structures used for storing and retrieving information.

        - **NOTE!!!!!!!!** For the scope of this course, we will focus only on numbers, Boolean and strings from primitive data types and on arrays from non-primitive linear data types. This means, we will instruct the computer to store, retrieve and process data in one of these data structure formats
            - It is crucial to master the other linear and non-linear data structures too, especially, if you decide to study machine learning. However, as a junior or mid-level web developer, it is very unlikely you are going to come across with non-linear data structures
- **Primitive data structure (data types) in JavaScript:** These data structures are the ones that are predefined by almost all programing languages. Whether it is JavaScript, Python or any other programing language, they know what these data types are. They know what to do with them. Ex: If you tell a programing language to add two integer values, it knows what to do with the integers. Primitive data types include:
    - **Number/integer**: which stores a range on mathematical integers
    - **String**: Strings are useful for holding data that can be represented in text form. Strings are written with quotes. You can use single or double quotes.
    - **Boolean**: stores logical values that are either true or false.
    - **Undefined**: The undefined datatype, whose value is undefined is used to denote an absence of a meaningful value. Do you remember when we said the datatype of a variable that has no value assigned is 'undefined', that is what we mean by undefined data type.
    - **Symbol**: was introduced in ECMAScript 2015, so just a few years ago. It's a very peculiar data type. Once you create a symbol, its value is kept private and for internal use. Symbol can be created using the factory function Symbol () which returns a Symbol. Every time you call the factory function, a new and unique symbol is created.
    - **Null**: JavaScript defines that null is an empty object value. Null represents the intentional absence of any object value. It is one of JavaScript's primitive values and is treated as falsy for Boolean operations.
    - **BigInt**: Bigint value represent integers with arbitrary precision. provides a way to represent whole numbers larger than 2^{53}-1. So, it means, with BigInts, you can safely store and operate on large integers even beyond the safe integer limit for Numbers.
- **Non-primitive data structure:** These are structures created by grouping primitive data together. Non-primitive data structure is a type of data structure that can store the data of

more than one type. The way they are grouped together varies depending on the kind of task we are trying to accomplish

- **Linear data structures**: If the non-primitive data are grouped in a sequential order, they are called Linear. Example:
  - **Array**: An array stores a collection of items in a manner that the position of each element can be calculated or retrieved easily by an index.
  - **Stack**: A stack stores a collection of items in the linear order that operations are applied. This structure is named as "stack" because it resembles a real-world stack — a stack of plates.
  - **Queue**: A queue stores a collection of items like a stack; however, the operation order can only be first in, first out. This structure is named as "queue" because it resembles a real-world queue — people waiting in a queue.
  - **Linked List**: A linked list stores a collection of items in a linear/sequential order. Hence, you must access data sequentially and random access is not possible.
- **Nonlinear data structure**: If the non-primitive data aren't grouped in a sequential order, they are called Nonlinear. Example:
  - **Tree**: A tree is a hierarchical structure where data is organized hierarchically and are linked together.
  - **Graph:** A graph stores a collection of items in a nonlinear fashion. Graphs are made up vertices, and lines that connect them, also known as edges. These are useful for representing real-world systems such as computer networks.
  - **Hash table**: A hash table, also known as a hash map, stores a collection of items in an associative array that plots keys to values. A hash table uses a hash function to convert an index into an array of buckets that contain the desired data item.

- Here is a good diagram showing the different types of Data Structures
  - https://cdn.ttgtmedia.com/rms/onlineimages/whatis-data_structure.png

2.3 Operators: arithmetic operators and assignment operators

- **Operators and Operands**:
  - **Operands**: The numbers (in an arithmetic operation) are called operands

    - **Operator**: Operator defines the operation that will be performed between the two operands
    - Example: 20 + 50 (20 and 50 are operands whereas (+) is the operator)
- **Assignment operator (=):** It is used to assign value or change value of a variable.
    - Example: var x = 10;
- **Addition assignment operator (+=):** This operator adds the value of the right operand to a variable and assigns the result to the variable.
    - **The syntax for (+=) operator is**: x += y   this means   x = x + y

```
Example:
let a = 2;
let b = 'hello';
a += 3; // means a = a +3, means a = 2+3. So, a
will be 5
b+= 'world'; // concatenation this means b= b +
“world”, means b= “hello” + “world”, so b will be
“hello world”
```

- **Subtraction assignment operator (-=):** This operator subtracts the value of the right operand from a variable and assigns the result to the variable.
    - **The syntax for (-=) operator is**: x -= y   this means   x = x - y

```
Example:
let a = 2;
a -= 3; // means a = a - 3, means a = 2 - 3. So, a
will be -1
```

- **Arithmetic operators (+), (-), (*), (/), (%) (++), (--)** : Arithmetic operators perform arithmetic on numbers, meaning, they work in a similar way we use them in math.
    - **The addition operator (+)**: adds numbers.

```
Example:
var x = 10;
var y = 5;
var z = x + y;
z = 15;
```

- **The subtraction operator (-)**: subtracts one number from another.

```
 Example:
var x = 10;
var y = 5;
var z = x - y;
z = 5;
```

- **The multiplication operator (*)**: multiplies numbers.

```
Example:

 var x = 10;
var y = 5;
var z = x * y;
z = 50;
```

- **The division operator (/)**: divides a number by another one.

```
Example:

  var x = 10;
 var y = 5;
 var z = x/ y;
 z = 2;
```

- **The modulus (remainder) operator (%)**: returns the remaining amount after dividing one operand by a second operand.

```
 Example:
var x = 10;
     var y = 4;
     var z = x %  y;
z = 2; // remainder is 2 because 10 = (4 * 2) + 2
```

- **The increment operator (++)**: increment (adds one to) its operand and returns the value before or after the increment.

```
Example:
  var x = 3;
  var y = x++;
  Output: "x:4,  y:3"
```

- **The decrement operator (--)**: decrement (subtracts one from) its operand and returns the value before or after the decrement.

```
Example:
var x = 3;
var y = x--;
Output: "x:2,  y:3"
```

2.4 Operators: string operators (concatenation) and comparison operators

- **String operator (concatenation operator) (+)**:
  - When using the (+) operator with string data type, the strings join or get concatenated. The (+) operator lets us add/join the contents of two or more strings together to create one larger string.
  - Note: The way you identify if a data is a string is by looking at the quotation marks around it.
    - Example: 2 is number, however, "2" is a is string
  - **Let's practice concatenation by using the (+) operator on strings**:

```
Example 1:
    var a = "Test Text";
    var b = "Another Text";
    var c = a + b;
    c becomes "Test TextAnother Text";

Example 2:
  var a = 4;
  var b = "Test";
  var c = a + b;
  c is now "4Test";
```

- **The addition assignment operator (+=) can also be used to add (concatenate) strings**:

```
Example:
  var myText = “seeYou”;
  myText += "Later";
  myText is now "seeYouLater";
```

- ○ **Comparison operators**: Comparison operators are used in logical statements to determine equality or difference between variables or values. Comparison operators always return a Boolean value of true or false.
    - **(==) Equals to:** This operator measures(checks) whether its two operands are equal, returning Boolean result .

```
Example:
var a = 5;
var b = "5";
a == b; // true, because (==)  it converts data type
to number and compares only values and both values are
equal here
```

- **(===) identity/strict equality:** This operator always checks the value and type of the two operands are the same or different, returning Boolean result.

```
Example:
  var a = 5;
  var b = "5";
  a === b; false, because (===) compares both value
and data type and the data type is not equal here
```

- **(! =) Not equal/ Inequality**: This operator tests inequality, to determine whether the values on either side of the operator are not equal. This operator is basically saying the two values on the left and right are not even loosely equal in value
- **(! ==) not identity/ Strictly unequal to with no type conversion**: This operator evaluates a strict inequality, which considers both the value and the type of the operands on either side of the operator. This operator is basically saying the two values are not equal in value or not equal in data type
- **(>): Greater than**: The greater than symbol in JavaScript may be familiar to you from math: >. This evaluates whether one value (on the left side of the expression) is greater than another value (on the right side of the expression). Like the == operator above, the greater than operator is not strict, and therefore will allow you to mix strings and numbers.

```
Example:
  var f = 72;
  var > 80; // output is false
  f > '30'; // output is true because the comparison
is not strict
```

- **(>=): Greater than or equal to**: This operator is typed as >= a kind of compound between greater than (>) and the equal sign (=).
- **(<) Less than**: The less than operator appears as the mirror version of the greater than operator. It evaluates whether one value (on the left side of the expression) is less than another value (on the right side of the expression). That means, like the

== operator above, the less than operator is not strict, and therefore will allow you to mix strings and numbers.

- **(<=) Less than or equal to**: The less than or equal operator will evaluate whether the value on the left side of the operator is less than or equal to the value on the right side.
- Here is the official ECMA Script definition regarding the rules JavaScript follows to handle comparison: http://www.ecma-international.org/ecma-262/5.1/#sec-11.8.5

- **General rules of comparison in JavaScript**:
  - Comparison operators return a Boolean value.
  - If comparing two numbers, it follows the standard numerical comparison.
  - If comparing two string data types, JavaScript uses the so-called "dictionary" or "lexicographical" order (ASCII). In other words, strings are compared letter-by-letter basis. This is handled by first converting the characters to their numerical equivalent representation.
    - If the values are of different data types, JavaScript tries to convert the non-number value to number value and tries to compare them. For example, if a number is compared to a string, JavaScript first tries to convert the string to a number value and,
    - If the string has a numerical value, JavaScript simply does numerical comparison between the two numbers.
      - Example: 2 < "12", the string "12" is first converted to number 12 and then 2 is compared to 12
    - If the string is a non-numeric string, JavaScript will convert this value to NaN (Not a Number). It then compares the original number with NaN. When number is compared to NaN it always returns false
      - Example: 2 < "Hello"
        - First, JS tries to convert the string "Hello" to number
        - Because "Hello" is a non-numeric string, it will be converted to NaN
        - Then JS compares 2 with NaN. This will return false because any number compared with NaN is false
    - If the string is an empty string, JavaScript will convert it to 0 and JavaScript simply does numerical comparison between the original number and 0.

        - Example: 2 < " "
            - JS will convert the empty string to 0
            - 2 is then compared to 0, in this case 2 < 0 returns false
    - **NOTE to come back and refer after we cover functions class**:
        - You can use methods like parseInt() and Number() to convert a string to number. These functions have their own rules of handling different scenarios.
            - For example: parseInt("2Abebe") returns the number 2. The remaining string "Abebe" gets ignored.
        - But even parseInt cannot convert everything to number.
            - For example: parseInt("Abe2be") is NaN. Hence, comparing NaN with any number would result in a false value

2.5 Operators: logical operators

- **Logical operators**:
    - Logical operators are used to determine the logic between variables or values.
    - We use these operators when we want to write conditional statements. We will learn about conditional statements later. For now, let's just focus on the operators.
- **There are three logical operators in JavaScript**:
    - **Logical AND (&&)**: This returns true if both operands are true. Meaning, if either one of the variables were initialized as false, the && expression would evaluate to false.

```
true && true = true
true && false = false
false && false = false
false && true = false
(1==1 && 2==2 && 3==7) // returns False
```

    - **Logical OR (| |)**: The OR operator, represented by two pipes, returns true if one operand is true

```
true || true = true
true || false = true
false || false = false
```

```
  false || true = true
(1==1 || 3==2 || 3 == 7) // returns True
```

- **Logical not (!)**: This returns true if operand is false and vice versa.

```
  ! False = True
  ! True = False
  ! (1==1 || 3==2) // returns False
```

- **Priority order when evaluating JavaScript operators:** Priority of operator (operator precedence) determines the grouping of terms in an expression and decides how an expression is evaluated.
  - Example: var x = 2 + 3 * 4; here, x is assigned 14, not 9 because operator * has a higher precedence than +, so it first gets multiplied with 3*4 and then adds into 2.
- **The following is a list showing operators that have the highest precedence on top**:
  - **Grouping**: ()
  - **Postfix incremen**t: … ++
  - **Postfix decrement**: … --
  - **Logical not**: !
  - **Comparison**: < , >= , = = = , !=
  - **Logical AND**: &&
  - **Logical OR**: | |
  - Refer this link from Mozilla MDN to see the priority order list for all JS operators:
    - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

2.6 Weak typing

- **Strong/Weak Typing:** is about how strictly data types are distinguished in a programming language (example, whether the language tries to do an implicit conversion from strings to numbers).
  - **Weak Typing:** Weakly typed languages do not require you to specify what type of data will be stored when you create a variable or an object. In weakly typed languages, operations between variables of different data types are legal because variables can be implicitly converted to other data types.

```
• Example of declaring variables in a weakly typed way
```

```
Example of declaring variables in a weakly typed way
  var someText = ‘Hello World’;
  var someNumber = 121;
```

- **Strong Typing:** In strongly typed languages, variables have definitive data types and operations between different data types are usually not permitted. Here strongly typed and require you to declare a variable’s type, such as int, float, Boolean, or String.

```
Example of declaring variables in a strongly typed way
String someText = ‘Hello World’;
int someNumber = 121;
```

- **Why is JavaScript a “weakly/loosely typed” language?** This is because JavaScript does not require us to specify a string is a string, or a number is a number when declaring a variable. JavaScript is smart enough to figure out what type of data you have and make the necessary adjustments so that you don't have to redefine your different types of data.

```
Example:
  var first = 123
  var second = "four"
  first + second // output is "123four". Because JS is
weakly typed, when adding a sting to a number, it
changes the number to a string and concatenates them.
```

2.7 Arrays: definition, declaration and adding values to an array, accessing array values with index

- **Array (definition)**: It is the most used data type from the nonprimitive data structures. An array is a special variable, which can hold more than one value. The values in an array are referred with a name and an index. Please note that an array can hold values that are of different data type.
- **Declaring/creating an array**: An array can hold many values under a single name and the below examples show how one can create an array and add values into the array.

```
Sytax to declare an array: var arrayName = [ ];
Example: var gradeResults = [ ];
```

- **Populating/adding values to an array**: You can populate an array by assigning values to its elements. Let's add values to the example array we declared above:

```
Sytax to add value into an array: arrayName = [item1, item2, ...];
Example: gradeResults = [55, 99, 73];
```

  - **Declaring and adding value to an array at the same time is also possible**:

```
var myCar = ["BMW", "Honda", "Ford"];
var x = ["TV", 99, "phone"];
```

- **Accessing array elements:** You access an array element by referring to the index number. Index is basically the location of an element of an array. In JavaScript, the first element of an array is at index 0.

```
Example:
var myCar = ["BMW", "Honda", "Ford"];
MyCar[0] // output will be BMW
[0] is the first element/ BMW
[1] is the second element/ Honda
[2] is the third element/ Ford
```

- **Changing array elements/values**: Changing the value/element of an array is as simple as assigning a value to an array. The example below will show how you can change the value of "Ford" to "Toyota" from the myCar array we created above:

```
var myCar = ["BMW", "Honda", "Ford"];
First identify the element you want to replace.
myCar[2]
Then, replace its value with the new value
myCar[2] = "Toyota"
```