JAVA

INDEX

❖ Table of Content

Git and GitHub

Git is like the Tech (version control system) which allows us to host our Projects on an online platform (as a Repository), where people from all around the world can contribute to it (or just our team from their remote servers). It hosts various feature like, which person contributed which feature change at what point of time and at what step during the project development. So if any of the change goes sideways, we can rollback to the last checkpoint. We can also work on the parallel process on the same project, and add all that at once to our project.

GitHub: GitHub is the Platform or software to host GIT (like others BitBucket and GitLab).

Cmd (Command Prompt): Command prompt basically allows us to change file structure in our system. Like creating, modifying, deleting a folder or file. Changing the contents of the file.

❖ Git Working →

Every change made to a project (every history/commit) is stored in another folder in Git called Git Repository, and its named as .git file.

After we have done {**git init**} (in a master project folder), every change made to that folder will automatically be captured and stored in a .git file. So, we don't have to explicitly capture any changes (like taking a photo of old and new script and file structure)

After performing any change within a project, it will be with RED MARK as an uncommitted change. Those can be checked by cmd {**git status**}.

→ For change to be recorded in history and to actually modify tit in the main project folder, first we must add that file to {**git add**} and then we must commit it (either by using GitHub UI way or by cmd {**git commit -m "[Descriptive message]"**}). [{git add} basically add the modified file to a "staged" area ready to be committed. And changes have to be staged first, in order for them to be committed]

We can check the status of every commit by cmd {**git log**}.

→ **Resetting Commit** :: Consider a scenario that we delete a file and committed it, now we did a bunch of further modifications to the whole project. Now suppose we want to restore the deleted file; we can't just remove it's commit from the middle (AS EVERY COMMIT IS BUILT ON TOP OF LAST ONE) and undo that commit. Instead, We can **un-stage** last commits till deleted file, and rollback to the commit which we want to restore to. Just copy the Hash code of the commit to which we want to revert to and un-stage it by cmd {**git reset hash_code**}. Now if you check {git log}, all the commits above it will be removed.

→ **Temporarily Commits** :: Another scenario, is that suppose, if we've modified bunch of files, but we don't want to commit it yet in the main project folder and we don't want to delete those requested commit as well. We can move them in git stash by {**git stash**}, such that it will not be shown in git status. Git stash is like a store room for all the requested commits, that we neither want to commit nor want to delete. After this cmd, this requested commits will not be shown in git status. {**git stash pop**} is used to bring back those staged or un-staged commit request to git status. And if we want to delete those commit request from stash area (store room), then we say {**git stash clear**}.

→ **Connecting Remote Repository to local Repository (local folder)** :: {**git remote add origin repo_url**} where repo_url is the link to the GitHub Repository. We can type in this command in working directory of the local folder in which we want to clone the repo. Now to pull the local changes to github repo by cmd {**git push origin master**} where master is the current branch. "Origin" depicts that the project is my own hosted and is in my account. If we have to add project hosted by some other client, then use {**git remote add upstream repo_url**}, where upstream depicts some other entity.

→ **Branches in github** :: There are several people working on a project on various feature. In essence every feature should have it's own branch to commit to. This is done in order to not affect the whole main project code, as the new feature being developed might have errors or bug in them, so it can affect the whole project. Instead, work on a separate branch for that feature. Make commit for that feature in that branch. And when the feature code is completed,

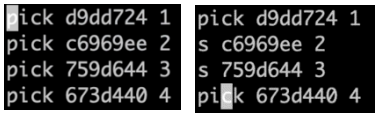
merge it with the main branch. To create new branch use cmd {**git branch branch_name**}. To move to the working branch, use cmd {**git checkout branch_name**} and make related commit to this branch; checkout will move head to selected branch. And to finally add the finalized branch code to the main project branch, use cmd {**git merge branch_name**}. People can then use this feature and can see the code of this feature in the main branch itself.

→ **Working with Projects** :: We cannot make any changes to the project hosted by other entity. In order to do so, first we have to **fork** it in our own account. Afterwards, we have to **clone** it in our local system by cmd {**git clone project_origin_url**}. All the project folder will be downloaded in our local system then. We can then make changes to this local folder and commit it on our own forked repo. After applying the necessary changes, if we want to make those changes in the project folder hosted by main owner. Then we have to sent "Pull Request" to the owner's account and to the required branch. He will first run and test those changes, maybe can ask for few modifications, then when finally accept our pull request. After all this process can only our changes will be visible in the main project folder. [Very Imp. – One branch can be associated with ONLY one pull request from a user. So must create new branch if want to ask for Pull request for multiple commits with different purpose. Otherwise, all the Commits will be requested within one PR only; and that can be troublesome for the testing purpose.]

If suppose by any chances you have added multiple different purpose commit to a single PR, then reset back (as described above) and then force push using {git push origin branch_name -f}. This will remove the extra commit before making a PR.

We can directly pull request all the updated commits form our forked branch using {**git pull upstream branch_name**}. If you want to fetch commits that have been done in the host repo but not in your fork repo, then either us the **fetch upstream** in github or use cmd {**git push origin branch_name**}.

→ If we want to merge multiple continuous commits into 1 commit, then we can use {**git rebase -i**} (-I is for the interactive cmd prompt). Change pick into s for the commits that you want to squash into the just upper commit node. For e.g. In side image, 2 and 3 commit will be squashed into the 1st commit.



Programming Language Basics

➢ Need of Programming Language

Machine only understands **0's** and **1's**. For us human, it is very difficult to instruct the computer in 0's and 1's. To solve this problem, we write the code in human understandable format known as **Programming Languages**. There are many programming languages available to us like C/C++, Java, Python and many more. All this programming language, first convert **Human understandable incrustations to 0's and 1's code**, which is called **Assembly Language**. This Assembly Language is then directly "read and executed" by the Machine's processor or CPU. The task of converting programming language to executable language is done by Programming Language's **Compiler**.

➢ Types of Languages

➔ Procedural, Functional, Object Oriented

- **Procedural** – procedural language is a set of linearly executed steps in order to run a program. Contains a systematic order of logic statements, functions, and commands to complete a task.
- **Functional** – Functional Language takes something as an input, which we called variables for that function, and gives us some new output without changing any of the variables. Used in scenarios, when we have to perform same operation a bunch of times on different input set.
- **Object Oriented** – OOP revolves around Objects. Objects has attributes (aka properties) and Functions (aka methods) associated with them. They are like the custom data type that we have built in order to make an analogous representation to the real world scenario or to any specific use case.
  For ex., Suppose we want to describe humans in our code somehow. "Humans" have few properties and functions related to them, but they can't be stored and coded separately. Like if we want to describe Human's different features and functions, we have to create many different variables and stored them individually and we have to do that for every human we are working with in our code. Now instead of this, we can create a **class** of human with many different features and functions associated with it, and we generate new **instance** of this class every time we want to deal with different human. We can store every human's features in their respective instantized class.
  Second e.g., Suppose we want to input something from an user while running the code. We can't code input method every time, we want to input something in our script. Rather we create a class of input method and call it every time we want to use it.
  This also helps in maintaining the stability of the code. Because suppose, we want to change some function code for a human object, we simply change its object code rather than changing every humans method.
  In short Object-Oriented makes it easier to develop. Debug, reuse, and maintain software.

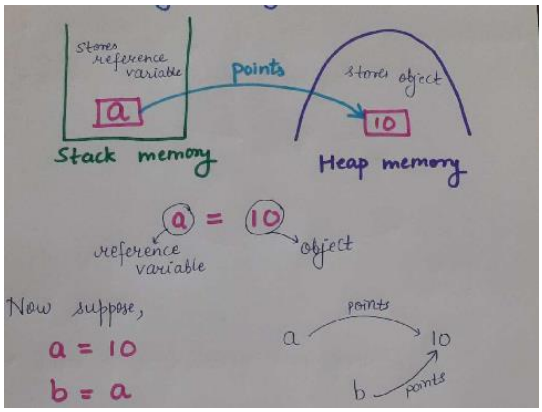
Java, C, C++, Python all this are Object-Oriented programming Language.

➔ Static vs Dynamic Language

Static	Dynamic
• Perform Type Checking at Compile Time • Errors will show at Compile Time • Must declare Datatype before we use it. • Gives us more control	• Perform Type Checking at Runtime. • Errors will show at Run time. • No need to declare datatype of variable. • Saves time in writing code but might give error at runtime.

As in dynamic programming, we don't declare datatype beforehand, we can change the datatype of the value the variable contain. Like 1st we can 'a' is 10, and later on we can say 'a' is 'Ashish'. It will work perfectly fine for Dynamic language like Python. Java is a Static Language.

➢ Memory Management (of Java only)

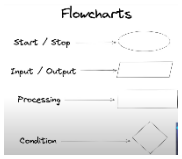


When we say suppose "a=10", then "a" is **reference variable** and "10" is the actual **object** stored in that reference variable. In the computer memory, "a" is stored in a stack fashion; and "10" is stored in a Heap management, where stack and heap are the form of **memory management** (we'll learn later). Now if we want to access the value or object 'a' contains, we will access 'a' first in the computer memory and see what address it is pointing to in the memory. Whatever address it is pointing to contains our desired object.

Note **1**. 'a' is not an object. The value it is pointing to is an **object**. **2** Another thing is that the object can be referenced by many different reference variables. For e.g. Name "Ashish" can be reference by my father, mother, teacher, manager. All this different reference variable are stored in stack memory and are pointing to same object 'Ashish' in heap structure. **3** If an object content is changed by any of the reference variable, then it will be modified for every of the reference variable. Because every reference variable points to same address in a memory. Java and Python follow this suite [Now we know, that in NumPy, when we say 'array a' = 'array b' and we make any modifications to a, it is also reflected in b. Because the object is what the array 'a' contains. And array 'b' is just pointing to it. That's why we create duplicate using '.copy' method. That way it creates a different object in a heap structure with different address just with same content.]

Garbage Collection → Garbage collection is the elimination process by Java to free up the memory. It happens when Object is **not** pointed by any reference variable. Example scenario, is when we change the content of the reference variable or when we delete the reference variable itself.

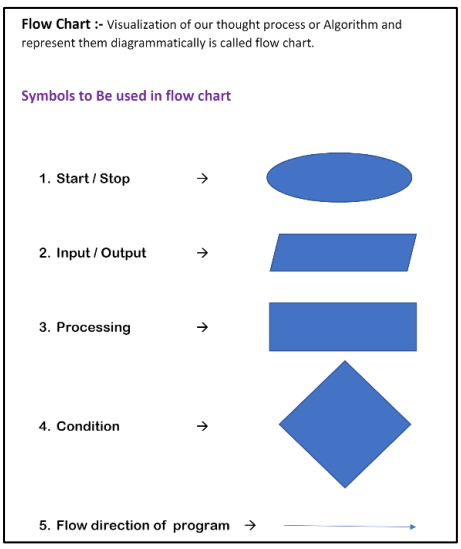
➢ Flowchart & Pseudocode

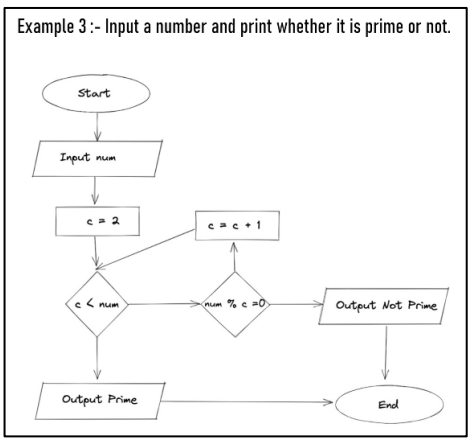


Flowchart → Flowchart describes the general **flow of the program** with all the logic. In other words, it is the logic flow of the program. In essence, flowchart and pseudocode is the one which is most important for solving any problem. Actual code can be written in any language, but first we have to describe its pseudocode and the flow to solve the problem.

Pseudocode → It is a way to write the steps of the code which is "**Human Readable**". We can call it **Human Readable Code**. It is meant for Human Understanding and not for machine reading. After writing Pseudocode (based on the flowchart or directly), we can then write the code in the Machine Language, with whatever Programming Language preferred. Pseudocode makes coding much easier.

Very Important Note → It is very hard to write Pseudocode and code, without building the flowchart first. We can do it without building flowchart, but it just gonna take double time.





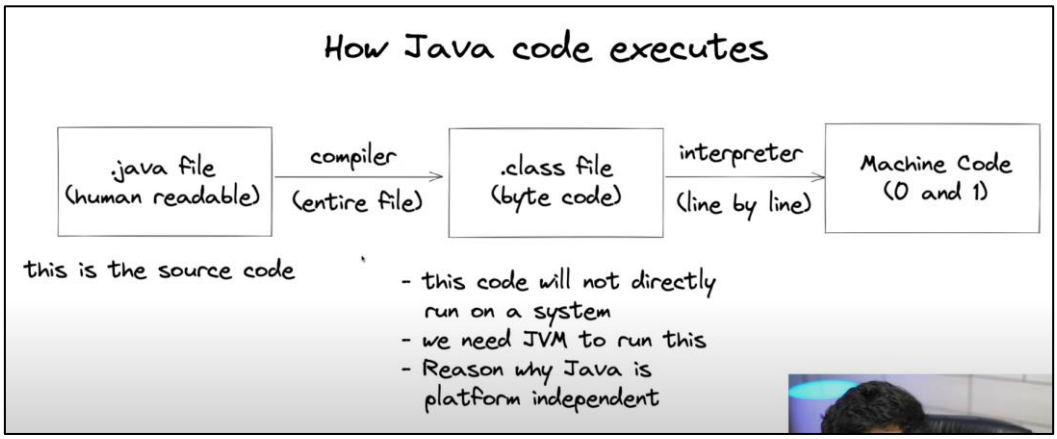
Pseudocode of Example 3

```
Start
Input num
if num ≤ 1
    print "Nither prime nor composite"
c = 2
while c < num
   if num % c = 0
        Output "Not Prime"
        Exit
   c = c+1
end while
Output "Prime"
Exit.
```

The order to solve any logic problem →

- Start with building a **flowchart based on logic flow** to solve the problem. (you'll definitely not arrive at solution flowchart in the first attempt)
- Build a **proper solution flowchart** (as many attempts it takes)
- Write a **pseudocode** (easily understandable by human) based on the flowchart.
- **Code the pseudocode**, with your desired programming language.
- Now optimize anything that you can in your flowchart and pseudocode. Remember → this is an important step and is done in order to reduce time and space complexity.

JAVA Language Theory and Architecture

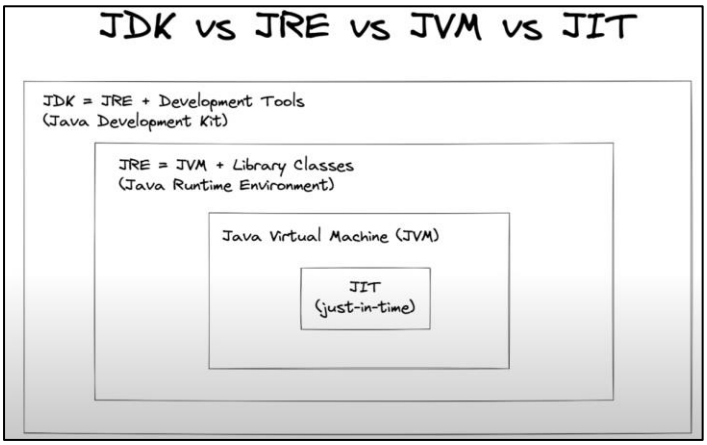


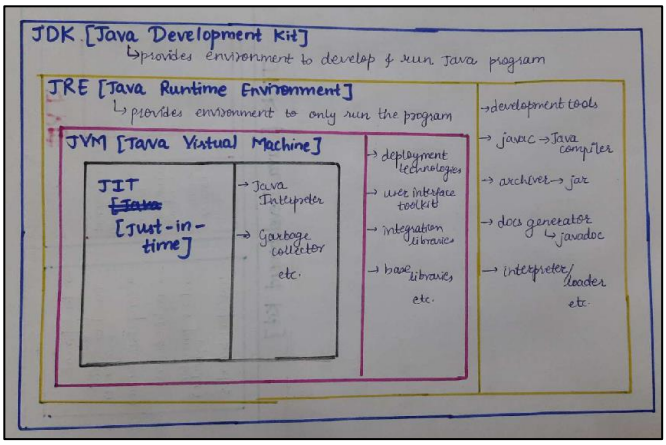
How Java code Executes → Java Compiler first transform **raw source code file (.java)** into something called **byte code file (.class)**. Now to transform this .class file into **machine readable format** we need something **called JVM "Java Virtual Machine"**, which will finally transform this Byte file into 0 and 1's. This middle step of byte code file does not exist in C/C++. One more thing to note is that Java is **platform independent**.

→ What do we mean by **platform independent**?

- **Byte code file can run on any "operating system"**. Meaning, if one has written a code in 'mac' then he can compile it and share the byte code file to someone with 'Linux system', and he can too run that in his system.
- In **C/C++ compiler** converts the raw source code file into .exe file which is directly machine interpretable and executable file. .exe is platform dependent. Meaning, executable file of windows will only execute and run on windows.
- In Java, we get bytecode file by the compiler, which needed JVM to convert it into machine interpretable file. Now this JVM is platform dependent (meaning OS dependent). But that problem is of OS and is not of any concern to us. We just need JVM for whatever our OS is.

➢ Java Architecture



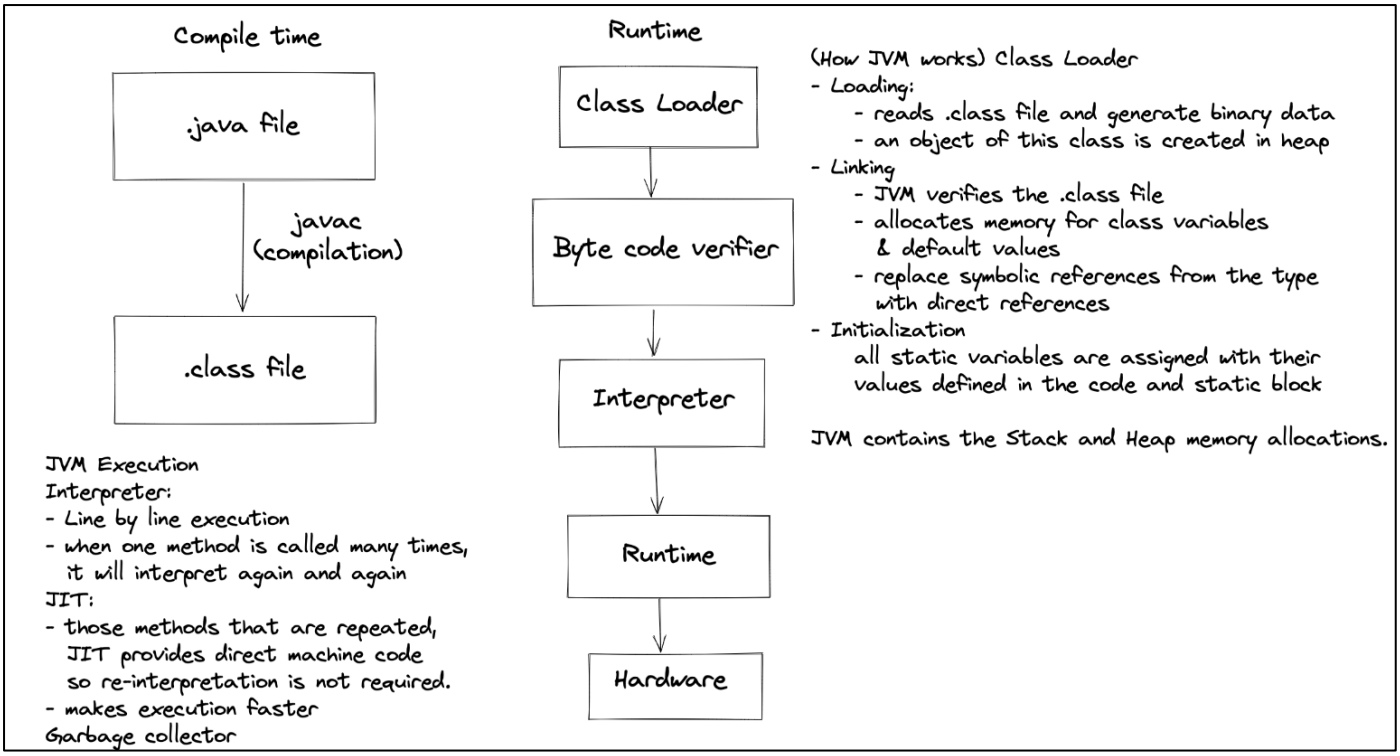


JDK

- Provides environment to develop and run the Java program
- It is a package that includes:
  1. development tools - to provide an environment to develop your program
  2. JRE - to execute your program
  3. a compiler - javac
  4. archiver - jar
  5. docs generator - javadoc
  6. interpreter / loader

JRE

- It is an installation package that provides environment to only run the program
- It consists of:
  1. Deployment technologies
  2. User interface toolkits
  3. Integration libraries
  4. Base libraries
  5. JVM
- After we get the .class file, the next things happen at runtime:
  1. Class loader loads all classes needed to execute the program.
  2. JVM sends code to Byte code verifier to check the format of code



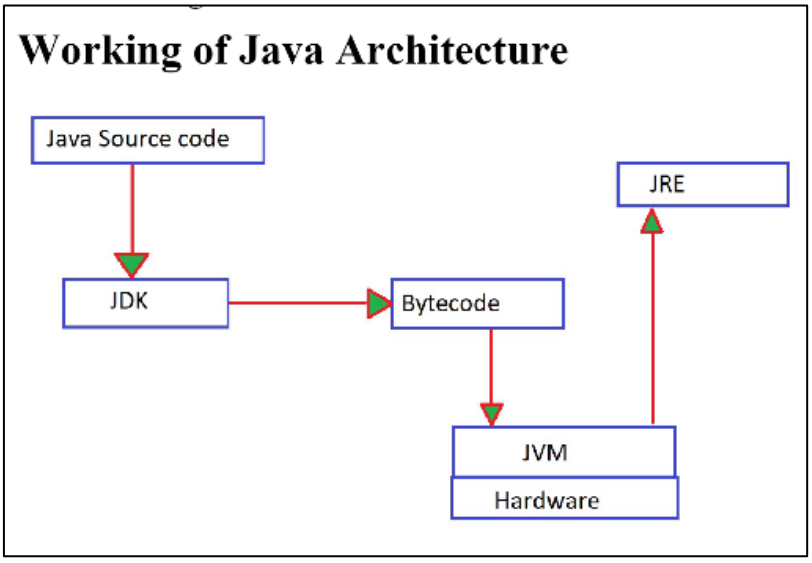
(How JVM works) Class Loader
- Loading:
  - reads .class file and generate binary data
  - an object of this class is created in heap
- Linking
  - JVM verifies the .class file
  - allocates memory for class variables & default values
  - replace symbolic references from the type with direct references
- Initialization
  all static variables are assigned with their values defined in the code and static block

JVM contains the Stack and Heap memory allocations.

JVM Execution
Interpreter:
- Line by line execution
- when one method is called many times, it will interpret again and again

JIT:
- those methods that are repeated, JIT provides direct machine code so re-interpretation is not required.
- makes execution faster

Garbage collector



JAVA PROGRAMMING

❖ **Conventions in Java** and **Basic Java code Structure**/ Important Points and Notes to Remember

```
public class Demo{
        public static void main(string[] args){
        }
}
```

1. ****Structure of Java File** → Source code that we write will be saved with "**.java**" extension. Every thing we write in java must be within classes. We can say that whatever we write in java is "**one big class.**" This 'class name' must be the same name as 'file name', in order for it to be executable. Class name which is same as file name must be the "**public class.**" A "**main function**" must be present within this public class. This main class is the executable code part of the java script. Anything written outside the Main function will not be executed.
2. **Everything we code in Java is in "Classes."** That means the whole script we write in a single .java file is one big class with class name as same as file name. Class is nothing but a combination of properties and functions. E.g., "main.java" file is one class file with class name "main". That means while writing the code in a file named "main.java", everything should be written inside this 'class Main'. [Special Note – Note that this is the case for the public class. If the class is not public then there is no need to put same name as file name, because then this class code is not the main code that will be shared to every piece of code from this specific file.]
   public class Main{//Code}
3. What is **Class** → Class is the name of the group of properties and functions. Every variable starting with **capital letter is a class**. We should also use this convention of writing first letter as Capital while declaring classes. [This is not compulsory, but it is a good practice].
4. **Public** is an "Access Modifier". It depicts that the code in this section is public to every piece of code, whether it be other classes, or other files, or other packages.
5. Every java code starts from the "**main function**." Here the name convention must be "Main." No other 'xyz' is allowed. Like the file name or the class name in the 1st point, we named it as Main, that is not a function but a class, and it can be named anything; it just had to be the same as the file name. But inside this 'xyz' class, executable code must be written inside the "function **main**." Otherwise, it will not start executing. We can define other 'functions' or 'classes as OOP', but they are not executable by themselves. They can be called inside this 'main' function, or can be written directly inside this 'main' function. In C/C++ also, executable code is written inside the function main.
6. What is "**Static**"? → It is a 'keyword' that allows the Main function to run without using any objects. As we can see above, main function (the starting point of executable code) is **actually** the function of the Demo class (for Demo.java file). Now, we also know that in order to run the object code, we first need to instantiate it or initialized it, in order to access its properties and methods. But here, we are not initializing the "Demo" class and we are running the "main" function without instantiating it. How to make this possible? From our previous understanding, what are the properties and functions called (aka variables and methods) which are not owned and depended by any specific object. Yes, we call them **static**. E.g., the variable 'populations of human' is a static variable for object class 'human', as it does not depend on it. That is why we declare the function "**main" as static** for the Demo class.
7. **Void** → Void is the return type of program. Since we do not want to return anything from the function "main". Main function is the actual executable code in itself and we are not returning anything from it. That's why we say the return type of main function is 'void'.
8. **Args** → args is basically a "**command line arguments**". If we want to pass params directly from command line.
9. **String[]** → String[] represents all the 'arguments' in a 'command line' and as an 'array' with the 'datatype string'. Suppose, we use args[0] in our code, and we pass one argument "hello world" in command line while executing; then this argument will be fetched by args[0] within the code.
10. **Command Line prompts** for running the code→ 1st step (compilation) -> **javac xyz.jav**
    2nd Step (execution) -> **java xyz args**

11. Afte compilation, Byte code file is created known as **".class file**" with the same name as our code file or class name. By convention, we store our source code in "src" folder and byte code file in "out" folder.
12. What are **packages**? → Packages are the complete project folder where our java files lie. They are created to generate and use some specified type of rules for our program. Like this functions or class can be used by the code in 'Ashish' package.
13. Java follows the **Unicode Principle**. So, we can put any language in Java. Like Hindi, Chinese.
14. VER VERY IMPORTANT & USEFUL → **Debug Feature**. Place the Debug Pointer at any line of code, and run with the debug mode. What will happen is that it will run the code line by line from the Debug Pointer. It helps us to know what is happening in the code line after line. We can place **Debug Pointer**, to start debugging from whichever line we want to.
15. Java does not have **Pointers** (unlike C/C++). It does not follow the concept of address.
16. (If IntelliJ IDE): Use 'CTRL + Click' to look within any java provided class.
17. ***Object Class** → Object class is present in the **java.lang** package. Every class in java is directly or indirectly derived from the **Object Class**. If a class does not extends other class then it is direct child class of Object class; And if class does extends any other class then it is indirect child class of Object Class. Therefore, the Object class methods are available to all of the Java Classes. Hence, Object class act as a root of inheritance in any Java Program.

❖ Basic Class & function in Java, provided by Java

```
public class Demo{
        public static void main(string[] args){
        System.out.println("Message");

        Scanner input = new Scanner(System.in);
        input.next();

        int a = 10;
        String name = input.nextLine();
        }
}
```

- **Object Instantiation** → **class_name ref_object_name = new class_name(params)**
  - **New** is a keyword that creates an object in java**.**
  - **Class_name** is basically an object name. It tells us which object we are trying to create instance of**.**
  - **"Class_name ref_object_name"** declares an object creation.
  - **"New class_name(params)"** actually creates an object in the heap memory.

- **System class** → Output {**System.out.println()**} ["sout" is the shortcut for IntelliJ IDE]
  - **System** is the class which gives us the basic method for giving output and other things.
  - **Out** is basically saying where the print function ought to give output. By default, it is set to 'Null'; Meaning the output will be shown on the console itself. We can set it to another File or terminal or any other place.
  - **Println** is the function to print and adds a new line at the end automatically.

- **Scanner class** → Input {**scanner input_object = new Scanner(input_source = 'System.in')**}

  {**Input_object.next()**}
  - **Sanner** is the class which allows us to take the **input** from any source. It is available in the "**java.util**" package.
  - To take the input, first we must **instantiate the object of the Scanner class** with whatever reference variable we want to call it to.

- In params/args of the scanner class, we must say the **source**. It is the InputStream source arg. i.e., from where we need to input the text. If it is from the system itself, then we say "**System.in**", if it is a file, we specify the file path.
- Note that this reference variable **input_object is pointing to an object** and not to any standard datatype. Meaning we can use all the **"properties and functions"** of the Scanner class with this "input_object."
- One such function of this scanner object is "**next(), nextInt(), nextLine()**", which says that store whatever the 'next text', 'next integer', 'next line' is typed on the console respectively. [Note – since we have used System.in, it will wait for input from the console.] "**Next**" represents that take the input from the next line in the currently executed console. Meaning we must give input, in the console, after it has reached the next() instruction. Suppose, if we have displayed some message in our code before the next() instruction, but we have given the input before reaching the display message command; then this input will not be read by the next() command.

❖ Data type

- **Primitive datatypes** are the one which cannot be break further. Like string can be break into char, so it is not a primitive. Int, char, bool are primitive datatypes.
- The way we declare variable in java is {**variable_dtype ref_variable = literal**}
- **Wrapper Class** → There are datatype wrapper class in java. It is basically "Object Class" of primitive datatype. Meaning primitive datatypes are wrapped in an object class. This is done to create and use many methods related to datatypes. e.g. "**Integer**" for "**int.**" In essence, they provide additional functionalities.

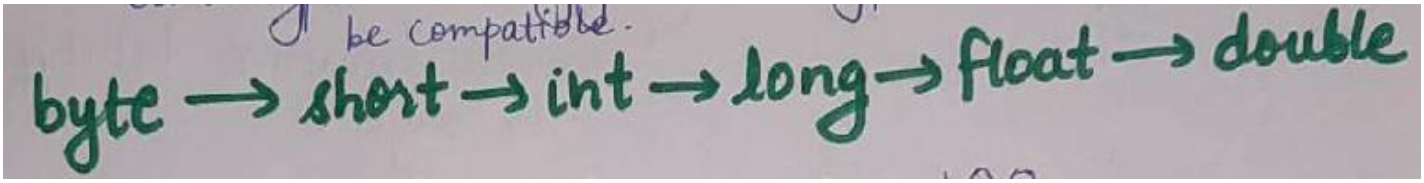
Data types	Description	Example
int	int is used to store numeric digits	int i = 26;
char	char is used to store character	char c = 'A';
float	flot is used to store floating point numbers	float f = 98.67f;
double	double is used to store larger decimal numbers	double d = 45676.58975 ;
long	long is used to store numeric digits which is not able to stored in int	long l = 15876954832558315l;
boolean	It only stores store t values i.e., true or false.	boolean b = false;

While initializing, in float and long we use 'f' and 'l' at the end, because otherwise it would consider those as "double" and "int" instead.

*Important remember point → When we denote String, we use "double quotes". And when we want to denote char, we use 'single quote'. This is important, because many a times, it gives type error. And it is different from other programming language habit such as Python.

Literals and Identifiers → Identifiers are the name of the reference variables, methods, class, packages, interface. Literals are the value and the object they point to. Ex:- a=10, a is 'identifier' and 10 is 'literal'.

Type conversion. This will happen automatically ONLY, if both are compatible AND destination type is greater than source. E.g. Int to Float is possible, but float to int is not possible. Look TpyeCasting.java for few important points.



Type Promotion → When performing a computation of different dtype or if range is exceeded, smaller dtype is promoted to higher dtype. It is known as **type promotion**.

Type Casting → We can **explicitly cast** one datatype object to another type. It is known as **type casting**. Remember, Type casting is necessary when we want to convert larger data type to smaller data type. In reverse direction, type promotion is automatically done. Still, it is a good habit to do casting wherever possible.

We type cast by **dtype ref_var = (dtype)value;**

Boolean → Boolean consists of **true** and **false** starting with small 't' and 'f'. Unlike other programming language, true **does not** corresponds to **1** and false does not corresponds to **0** in java.

****Refer to com.basic folder for codes.**

Conditional and Loops

- For conditional statements and loops, every programming language has the **same code structure**.
- **C and Java follows the same syntax**. In Python, we do not need curly braces, simple indentations after the condition are sufficient.
- In any programming language, **condition comes within "Normal Braces"**. And **Code Block comes within "Curly Braces."** Python only need indentation for the code block.

❖ Conditional Statements →

Conditionals are simple **If-Else** statement. Execution pointer enters the Code block for which of the logic condition is met. They create separate branch in Flow chart for each of the "Logic Condition." We can use only the if statement. We can use if-else statement. We can use if-elseif-else statement for multiple conditions.

There is also **Switch** statement. Switch compares the expression value with multiple cases and execute code block in which case matches the expression. Case must be the constant or literal. Default will run if none of the case matches the expression. Break is used in each case statement to break out of the switch sequence. If break is not used then it will move on to the next case (no matter the condition of further cases). Duplicate cases are not allowed.

1. **Only If**
   If(condition){code block}
2. **If-Else**
   If(condition){//Code Block if T}
   Else{//code block if F}
3. **If-elseif-else**
   If(condition){//code block if T}
   Elseif(condition){//code block if T}
   Else{//code block if none of the above is T}
4. **Switch(Expression){**
   Case 1{//code; break;}
   Case 2{{//code; break;}
   Default{{//code;}}

❖Loops →

Loops run the same "**Code Block**" multiple number of times, with new incremented or decremented value of the **iterator variable**. Execution process **jump** back to the starting point of code block (code block within the loop) with new value of the iterator variable in the flowchart.

There are 2 loops system. "**for"** loop and "**while"** loop. Use "for" loop when the number of iterations is known. Use "While" loop when the number of iterations is unknown.

There is one more loop called "**do-while**" loop. The only major difference of the "do-while" loop with while loop is that, the code block within the do-while loop is going to execute first and check for the condition later in while statement for further iteration of the loop. That also means that the code block is going to execute at least once, no matter the condition of the while statement.

While Loop	Do while loop
→ used when no. of iteration is not fixed	→ used when we want to execute the statement at least ones
→ Entry controlled loop	→ Exit controlled loop
→ no semicolon required at the end of while (condition)	→ semicolon is required at the end of while (condition)

1. **For(initialization, condition, increment/decrement)**
   {code block}
2. **While(condition)**
   {code block
   Increment/Decrement}
3. **Do{**
   Code Block
   Increment/Decrement
   }while(condition)

****Refer to com.logic folder for codes.**

In Code Practice, we have performed 5 questions for Logic statement and loops practice.

- **Maximum Finder**
- **Lowercase or Uppercase**
- **Fibonacci Sequence**
- **Counting Occurrence of digit in large number.**
- **Reverse the Number**
- **Lastly, we have built the basic calculator.**

Functions (via Java)

Important Points regarding the "**Functions**" via Java.

- Functions are the block of code that you do not run directly but instead **call it in the main function for execution**.
- Functions are the **steps of internal computation** that "requires some input" and either 'return some output' or 'display something within the function code itself.'
- We **use functions** when we need to repeat some steps of computational process **again and again** in the main function. To write the same code lines again and again is not 'Efficient'. That is why we make the function for those code lines, and call this function with just the "function name" whenever we need to use it. We can call this function 10 or 100 times just by its name.
- We call the function by {**function_name (args/params)**}
- If we return some output from the function, we need to declare **return type** of the output while declaring the function.
- We must declare **access modifier** type of the function from "Public", "Private", "Protected", "default."
- We need to declare whether the function is **static or not**.
- Unlike Python, in which we can pass "params" (or arguments) with the declared reference variable name. In Java it is not possible. Furthermore, if we cannot use declared ref variable in function calling, meaning we can't swap the params. That is to say, we must pass the arguments in the specified order as declared in function. Like{**sum(a=30,b=20)** OR **sum(b=20,a=30)** OR **sum(a=num1, b=num2)** OR **sum(b=num2, a =num1**) all these methods are not possible. Only correct way is {**sum(30,20)** OR **sum(num1,num2)**} [Special note on these, when we enter the params value onto the function call within the "IntelliJ idea", it automatically inserts and highlight which variables are which. That is not the Java code feature but useful feature by the "Intellij IDE" itself.]
- Important → **Main Function will not include anything from the Function call's internal Process.** Any objects, variables created inside the function will be flushed after the function is done with his task. The only thing that Main function includes from the function call is its return value; And the Message Displayed within the function call.
- If we need to use the Function in another static function, then we must declare this function also as static.
- We can declare function in and out of the main function. But we cannot declare the function outside the scope of the "Main Class."
- Any line of code after the **return xyz** within the function will not run. The function ends executing after it sees return. Return statement transfer the program controller back to the caller of the method.

```
Access_modifier   static   return_type   function_name(params) {
        Code block
        Return xyz;
}
```

***Java follows "Pass by Value" and not the "Pass by Reference"** → When we pass the variable to the Function as a param, function creates its own reference variable (even though the name from function and the main is same) that initially points to the same object (literal value) in the memory. Now when the object value of ref variable from function code is changed, it creates a new object with new value. And the reference variable from function now points to this new object value created.

This means the original object value of the reference value from Main is unchanged. To change the object value from the Main, we have to "Return" the reference variable with new Object Value from function call.

An example of when the object value of Reference value from Main is changed from the function code itself → When we pass the array into the function. And we change the specific index value, then the change is also reflected in the original Object from Main. This is because at this time, we have not changed the entire object, and we have modified part of Object value from the original position in the memory. Therefore, no need to create new object. In Java, we do not have pointers. We not get address or anything in java.

❖ Some Extra Topics/Lessons related to 'Functions'

➢ Scoping

Function Scope → Anything defined in a function will remain in a function. It cannot be accessed outside the function or outside the scope of a function.

Block Scope → Anything initialized in a block is accessible inside the block only. This block can access thing from outside the block and are modifiable. But things instantized here is not accessible and modifiable outside the block. To access modifiable thing from the block, instantized it outside the block.

Loop Scope → Anything initializes inside a loop is accessible only inside the loop code block.

In short, very-very important point to remember for scoping

- Anything already initialized cannot be initialized again. (Only exception is **Shadowing**)
- Anything initialized outside is accessible inside the code block.
- Anything initialized inside the code block is not accessible outside it.
- To use anything Initialized inside the code block, Re-initialize it outside the code block and AFTER the code block.

➢ **Shadowing** → Any variable declared and initialized outside the MAIN function, can be declared and initialized again inside the MAIN function. The object value of the ref variable is taken from outside the MAIN until it was declared again in the inside scope. After declaring it inside the lower-level scope, the value at higher level scope is "**shadowed**" and hidden by this newly initialized object value. From here on, the value for this same reference variable will be from this lower-level initialization. Shadowing not take place for the methods.

```
public class basic_function2 {

    static int x = 50; // Shadowing  2 usages
    public static void main(String[] args) {

        System.out.println(x); // Will print 50
        int x=10;
        System.out.println(x); // Will print 10 (Shadowed)
        temp();                // Will print 50
    }
    static void temp(){  1 usage
        System.out.println(x);
    }
}
```

➢ VarArgs (Variable number of Argument/Params)

If we do not know, how many arguments we are going to pass in a function call, then we can use something called VarArgs in the function declaration. It simply works by declaring {**fun_name(dtype …ref_var_name)**} E.g. **max(int …num)**. What it will do is it takes **ANY** number of params in a function call and pass it as an **ARRAY** for the function computation.

```
public class VarArgs {
    public static void main(String[] args) {
        UnknownArgs( ...v: 51,35,42,0,56);
    }

    static void UnknownArgs(int ...v){  1 usage
        System.out.println(Arrays.toString(v));
    }
}
```

Note – vararg parameter should be at the end of the list. Because java would not be able to know when the vararg parameter will start.

➢ Overloading/ Overriding

- Two or more than two function can EXIST, if their required argument parameters are different.
- While function calling in the Main, based on the args passed, java knows which function to call. This is called function overloading.
- This happens at Compile time in Java.
- Two things should be kept in mind. Either the number of Argument should be different. Or the type of the Argument should be different.

```
public class Overloading {
    public static void main(String[] args) {

        message( a: 50);
        message( b: '5');
        message( a: 50, b: 100);

    }

    static void message(int a){  1 usage
        System.out.println(a);
    }

    static void message(char b){  1 usage
        System.out.println(b);
    }
    static void message(int a, int b){  1 usage
        System.out.println(a+b);
    }
}
```

****Refer to com.Functions folder for codes.**

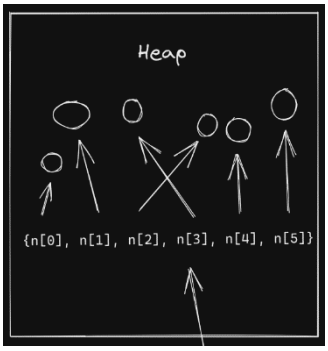
Arrays

➢ **Important Points** regarding "**Arrays**" via Java.

- **Arrays** are used to store "**collection of data**" with similar data types in a "single object" that is accessible by single reference variable and its indexing.
- **Need of an Array** → We need array when we need to store multiple values within a single object. It also makes computation lot easier of that data.
- Arrays are unlike any other **Object** in java. Hence need to initialized as such. {**dtype[] ref_var = new dtype[]**}
- "**Square parentheses[]**" identifies that this is an Array object. We do not say the object name as an Arr. Instead, we mention the **datatype of the array** for the object creation. It tells us about the nature of the elements within the Array.
- All the elements within the Array object of Java MUST be of the same type.
- "**dtype[] ref_var**" declares an array with the name "ref_var", in the stack memory.
- "**ref_var = new dtype[size]**" initializes an array and actually create the array object in the heap memory, with the size of "size" elements.
- We can also declare and initialize an array directly by {**dtype[] ref_var = {value 1, value2, …, value n}**}
- For integer array, java initializes and set every element to be "**0**". And for String Array, it set every element to "**Null.**"
- Each element of an Array is itself an object.
- To print Array in an convenient manner, we use "**Arrays.toString(arr_name)**"

Array memory management

- Initializing an array creates the object in memory at "Runtime." It is called "**Dynamic Memory allocation.**"
- As we know Java does not have the concept of pointer so java has another method of storing array. Heap does not follow "**Continuous memory management.**" And all the object created are stored in Heap memory. Meaning all the elements of array may not be continuous. [It depends on JVM].
- Every element is stored in random memory allocation in Heap memory. Every element in an Array has its own reference variable in JVM, which are used to call them from the heap memory.



➔ **For-Each Loop** → We can declare for loop for Array Elements in a special way called **For-Each Loop**. In this way we do not have to describe for loop with conditions and increments as such. We can directly for each element in array arr_name, perform the specified task. Way to declare for-each loop

- **For(dtype ref_var: new dtype[]{elements}) {//code block}**
- **For(dtype ref_var: arr_name) {//code block}**

➢ **2d Array or Multi Dimension Array**

- **2d Array**. 2d Arrays are just like Matrix. We can declare 2d Array with 2 square parentheses. First for the number of row and second for the number of columns.
- Each row in the multi-dimensional array is separate object in itself. That means, it can have its own **separate dimension** and it will work.
- We cannot insert values of every element in Array directly after initializing it. Meaning, if we want to assign value to every element in Array in one go, we need to do so while initializing it or declaring it. Otherwise, we need to use 'for loop' to assign values to each element.

➢ ArrayList

- ArrayList are used when we **don't know the size of array** we are gonna needed. Rest ArrayList are the same as array. [This is same as vector<> from the cpp programing.]
- Syntax for ArrayList: {**ArrayList<dtype_wrapper_class> ref_var = new ArrayList<>(initialSize)**}
- **For the data type in the Array List, we cannot insert primitive Data type. We need to use **Wrapper class** of the needed data type. Like "Integer" for int.
- ****VERY IMPORTANT** → We have to use "**Method**" for performing any operations with ArrayList. This is completely different from the Array. Like setting the value, adding the value, getting the value, removing the value.
- Some **commonly used methods** are :: "add()," "get(index)," "set(index, value)," "remove(index)."
- ArrayList are a part of "java.util" package.
- We can also create **multi dimension ArrayList** (if unknown size)
- Because of its unknown size functionality, it is slower than standard array.

How ArrayList works internally → First, we create an ArrayList with some initial size. When we fill an ArrayList by some amount, then internally, JVM will create new ArrayList with say, double the size. And copy all the elements from this old ArrayList to this new ArrayList. Old ArrayList is now deleted.

****Refer to com.Arrays folder for codes.**

Check for the Input and Output for "Array, 2d-Array, ArrayList, 2d_ArrayList" example from the Code.

Questions → 1. Maximum, 2. Reverse and 3. Swap using Arrays.

Personnel real life observation (for the requirement of Data Structures)

Necessity of Array or any Linear Data Structure.

- As we were solving the basic problems of maximum from 10 numbers, Fibonacci sequence, reverse and many more, we come across the problem of storing every input in different temp ref variable.
- Now to call this variable every time we have to use explicitly temp ref variable name.
- This caused a problem in real life.
- Now what if we store this in an array or any other Linearly ordered Data Structure.
- We can then call them by just using the **indexing position** rather than by every ref variable name (inside the loops, logic and print statements, etc.).
- So you see how easy life gets for solving the problem just by the inclusion of the Data Structure, especially Array.
- This also solves the problem of creating redundant temp reference variable for every value.
- That is why different Data Structures are build and used to solve the problems.

That is one way to solve Data Structures problem. Look at the problem, figure out a vague solution, and see what type of Data Structure can make this Solution solvable, easy, and efficient.

String

```java
// String are Immutabel. Pass by Value (and not by reference)

String name = "Ashish Agarwal";
System.out.println(name);  // Will return 'Ashish Agarwal'.

// Strings are immutable.
//   SO when we reassign anything to string object,
//   it will create new object value and then point to the newly created object.
//   Earlier created object value "Ashish Agarwal" is still there in the heap memory, but now not pointed by any ref var.
name = "Ashish";
System.out.println(name);  // Will return 'Ashish'.

// If new ref var object created and stored the value of the object that is already created,
//  then both will point to same object value.
String naam = "Ashish";
// Both 'name' and 'naam' pointing to same object value "Ashish" in the heap memory.
```

- Strings are a "**sequence of characters**".
- It is initialized in **double quotes**.
- It is **non-primitive datatype**.
- It is **immutable**. Why → For security purpose.
- It is declared and initialized like any other datatype. **String ref_var = "lorem ipsum";**
- We can get the characters of string using method called "**string_name.charAt(index)**"
- ****Always follow "**Pretty Printing**" in competitive coding.

➢ String Pool

Whenever we create a new String object with object value that already exist in the heap memory, then it will not create new Object value and instead point to already existing one. This is known as **String Pooling**.

String follows Pass by Value concept of Java.

When we reassign string object to new value, then why it got changed? → It does not change. It just creates new object value and points to this newly created object value. (**Pass by Value** concept of Java). We can verify this concept using the "**==**" operator and "**.equals**" operator.

```java
        // How to compare object value and at same time compare their memory allocation.

//        Comparison operator "==" checks for both the 'content' and their 'memory allocation'.
//        If both the comparison operand are pointing to same object value, then only "==" will return true.
        String name1 = "Ashish";
        String name2 = "Ashish";
        System.out.println(name1 == name2);  // Return 'True', because both the content and address are same.

        String a = new String( original: "Ashish");
        String b = new String( original: "Ashish");
        System.out.println(a == b);          // Return 'False', because content are same but not the memory address.

//        ".equals" operator only check for the "content" and "not the memory allocation".
//        Suppose if we create new operand with '.copy', it creates new object value with same content.
//        In those cases "==" return false BUT ".equals" return true.
        System.out.println(a.equals(b));
        // Return 'True', because content are same. And .equals does not care for address
```

➢ String Concatenation → with "+" operator

- We can concatenate the texts with **operator "+"**.
- When an integer is concatenated with "+", it is converted to its wrapper class "Integer".
- Texts are concatenated if and only if, 1. They are "primitive"; 2. Either of them is "String".
- Class Objects can not be concatenated, if both of the operands are class objects.
- We can modify the role of "+" operator in C++ or Python, which is called "operator overloading". But java does not support this due to software engineering concerns.

➢ String Builder Class

StringBuffer main advantage is that, it can 'store' and 'add' any **unknown size** of "sequence of chars". It's internal working is just like that of ArrayList for Arrays. It is mutable.

We can add sequence of characters in the same string Object using **".append**" method. It will add to the same String Object in the memory instead of creating new.

➢ String Methods

➔ **String** and **its related Class** provide many of the useful **METHOD** for us

- Length(), toLowerCase(), indexOf(), .lastIndexOf(), .strip(), .split(), and many-many more.

****Refer to com.Strings folder for codes.**

For question we have solved **Palindrome Program, Random String Generator.**

StringBuffer, BigInteger, BigDecimal

➢ String Buffer class

StringBuffer is mostly same as StringBuilder. **StringBuffer** main advantage is that, it can 'store' and 'add' any **unknown size** of "sequence of chars". It's internal working is just like that of ArrayList for Arrays.

- Sometimes, rather than a small input, company asks us to take the input file which contains many lines of input. And it can be of unknown and large size.
- StringBuffer is "**mutable**". It is Efficient (same as the case for StringBuilder, i.e. no new object creation).
- StringBuffer is "Thread safe" (meaning, if one thread is working with the data, another thread can't work on it.) This is not the case with StringBuilder.

StringBuffer has almost same method as String and StringBuilder.

➢ BigInteger (Large Integers) & BigDecimal (Large Decimals)

How to work with large integers.

- As we have ArrayList for arrays, StringBuffer for String, we have **BigInteger** for Integers.
- It is a class in the "**java.math**" package.
- Similarly, we have **BigDecimal** for large Decimal number.

File Handling

- We have to understand "**Streams**" for understanding file handling.
  - **Stream** → Stream is basically the "**Sequence of Data**". Like we say water stream. It is a **data stream**, where data is of two types "**byte**" or "**character** (Unicode)"
  - Java implements these within class hierarchies in "**java.io**" package.
  - There are two types of Streams. "**Byte Stream**" and "**Character Stream**".
  - The two of the most important method for all the classes within these java.io hierarchies are **read()** and **write()**.

Byte Stream and Character Stream

- **Byte Stream** is when we want to read or write streams in "byte format or **binary data format**". It is useful for like when we have media files (audio, image, etc.), or File Stream, etc.
- It has main two top level classes hierarchies → **"InputStream" (System.in)** and **"OutputStream" (System.out).** Each of these have several concrete subclasses.
- For a byte stream, we can only insert characters within 256 limit of ASCII characters.

- **Character Stream** is when we want to read or write in "sequence of **character format**". Like reading Unicode or ascii streams.
- It has main two top level classes hierarchies → **"Reader" (read())** and **"Writer" (write()).** Each of these have several concrete subclasses.
- For character stream, as it follows Unicode, we can insert whatever character from whichever source we want to and it will work. It does not have a limitation of 256 characters only. But it won't work with image, audio files as such, because they are in byte format.

IOException

- **Important → For every task performed of IO or Input output with file or java.io classes, we **HAVE TO** include **IOException** with "**try and catch** functionality." It basically informs us of corrupt file, file not found, not able to read and such errors. It is included in java.io package, so we must import it too.

- There are some **pre-defined streams** in java.io.
  - System.out → Standard output stream. (Console)
  - System.in → Standard input stream. (Keyboard/Console)
  - System.er → Standard error stream. (Console)

- Some of the **Input Class** Hierarchies in java.io Package

  For **Input** (Note → Every class has read() method, that reads one byte at a time.)
  - First, we have **InputStreamReader** class (EXTENDS Reader). It reads text from Byte Stream and converts it into character stream.
  - We have **FileReader** class (EXTENDS InputStreamReader EXTENDS Reader). It reads characters from file (using file path). It reads character-based file.
  - We have **BufferedReader** class (EXTENDS Reader). It reads text from Character stream. It also has an additional method called **readLine**(), which reads the whole line at one go; this is a huge advantage. We can read byte-based stream from BufferedReader using BufferedReader(new InputStreamReader(InputStream)).
  - For InputStream, we can either use System.in or File.

```
Object -> Reader -> InputStreamReader -> FileReader
```

- Some of the **Output Class** Hierarchies in java.io Package

  For **output** (Note → Every class has write() method, that writes one byte at a time.)
  - First, we have OutputStreamWriter class (EXTENDS Writer). It writes text in Character Stream format.
  - We have FileWriter class (EXTENDS OutputStreamWriter). It writes text to given file (using file path). It writes the file in character-based format.
  - We have BufferedWriter class (EXTENDS Writer). It writes text in Character stream. It has an additional method called newline(). With these we don't have to explicitly mention "\n" to end the line.

```
Object -> Writer -> OutputStreamWriter -> FileWriter
```

```java
// "System.in" is "InputStream"
// InputStreamReader takes the input in ByteStream and gives output in Character Stream.
try (InputStreamReader isr = new InputStreamReader(System.in)){

    System.out.println("Enter Some number");
    //InputStreamReader.read() will read one character at a time.
    int letters = isr.read();  //read() method returns an integer
    // Even though, it will read one character at a time, we can still provide all the characters in one go and use it.
    // For detailed explanation, read below comment blocks.
    /*...*/
    //ISR.read() methods check whether Stream Object is ready to read or not.
    // Basically, whether it is ready or not.
    while(isr.ready()){
        System.out.println((char) letters);
        letters = isr.read();
    }
    /*
    See -> In the above while loop, when we've print the first character (from our input stream),
    it again read the next character from ALREADY ENTERED input stream from the first call.
    Now, notice the difference here. We've already entered the Input stream from the first call.
    If it was the scanner. We've to type the input again and at this point of time.

    **This way we can type in all the Inputs in one go and read the input stream whenever required.
     */

    // We have to close the InputStreamReader. Because it is using the "Resource".
    // Though after ver 7, try and catch will automatically close it.
    isr.close();
} catch(IOException e){
    System.out.println(e.getMessage());  // It's a code to print IOException message.
}
```

****Refer to com.File folder for codes.**
