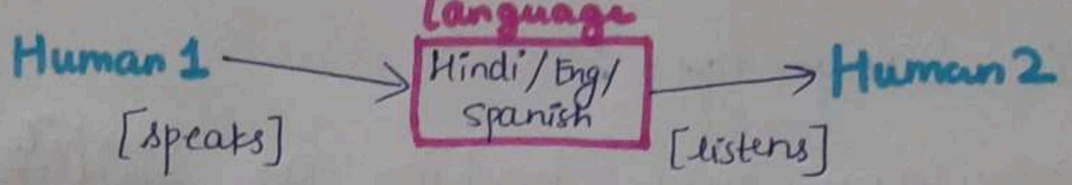
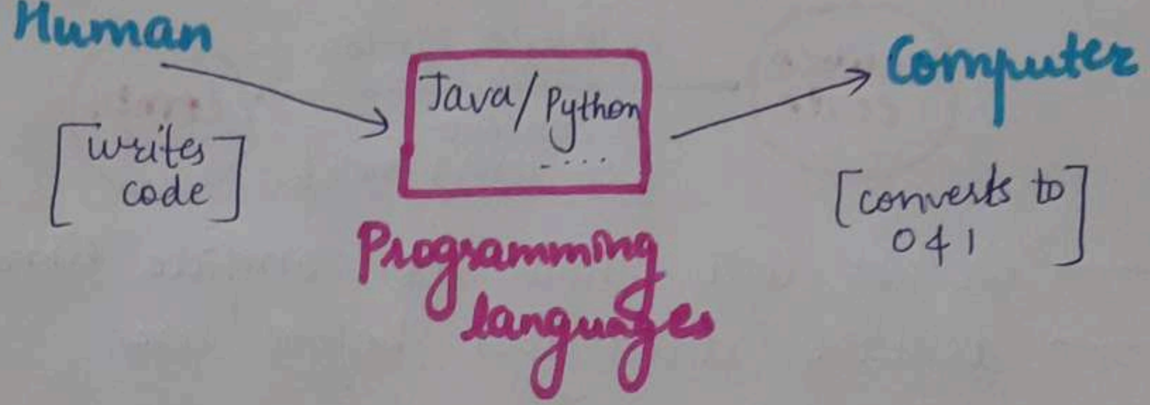
2|8|21

Introduction to Programming Language

- Computers at very minute level only understands zeros & one's (0's & 1's)
- What is programming language?





- Types of Programming Languages:

Procedural:

→ series of well-structured steps & procedures to compose a program

→ contains a systematic order of statements functions and commands to complete a task.

Functional:

→ Writing a program only in pure functions i.e., never modify variables but only create new ones as an output

→ Used in a situation where we have to perform lots of different operations on the same set of data like ML.

Object Oriented:

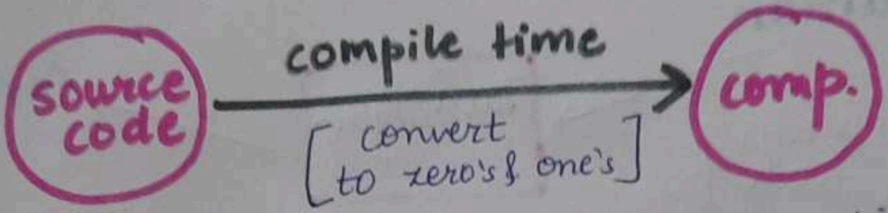
→ Revolves around objects

→ code + data = object

→ developed to make it easier to develop, debug, reuse & maintain.

• Static Languages:

→ Perform type checking at ~~runtime~~ compile time.



→ errors will show at compile time

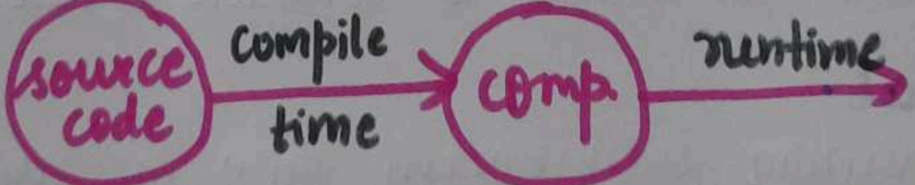
→ declare datatypes before use

int a = 10

→ More control over the program.

• Dynamic languages:

→ Performs type checking at runtime



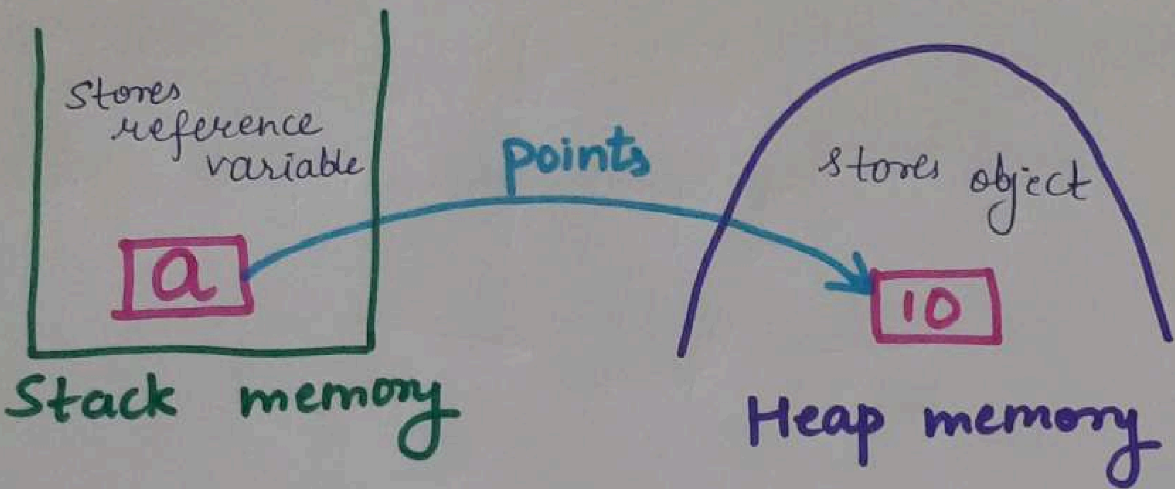
→ error might not show till programs run

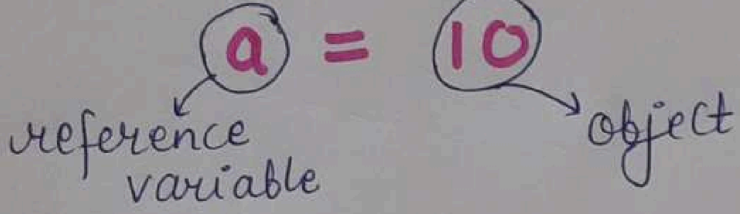
→ no need to declare datatype of variables

a=10 [language by itself figures out data type]

→ Saves time in writing code but might give error at runtime.

→ Memory Management :

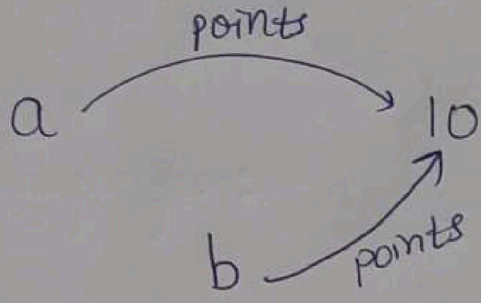




Now suppose,

a = 10

b = a



→ more than one reference variable can point towards one object.

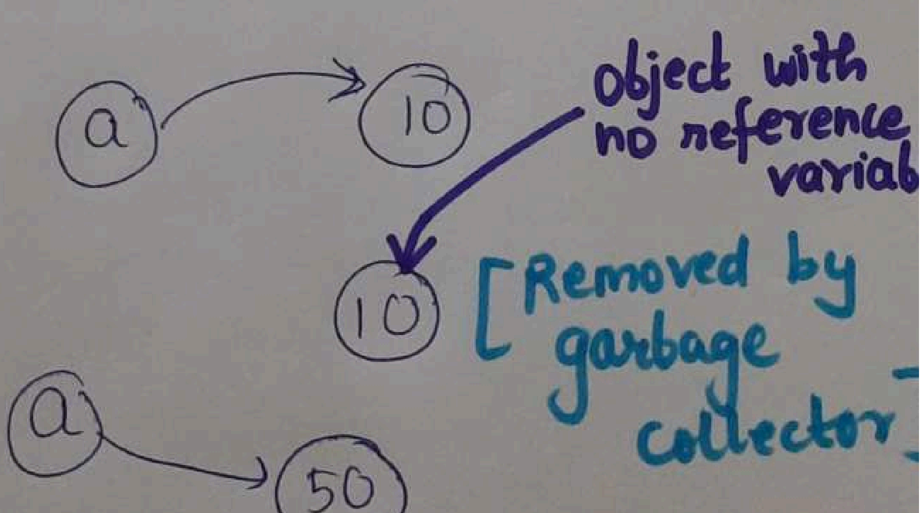
→ If any of the reference variable changes the object then it is changed for all reference variable ~~poi~~ that points towards same object.

Now intially,

a = 10

then,

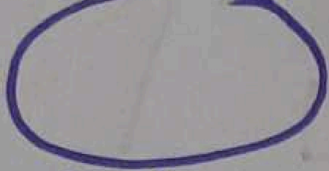
a = 50


a
10
object with no reference variab
10
[Removed by garbage collector]
a
50

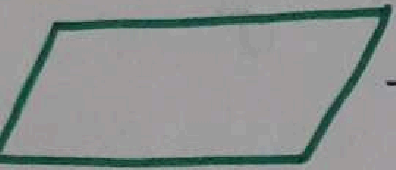
2/8/21

Flow of program

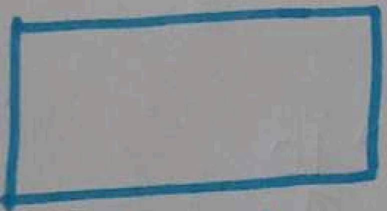
★ Flow Chart Symbols :



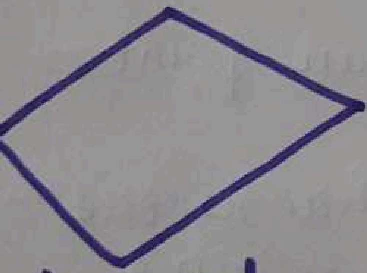
→ **Start/ Stop** → Represents start or end point of program.



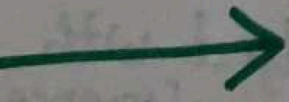
→ **Input/ Output** → Represents the input & output



→ **Processing** → Represents a process like addition, subtraction et

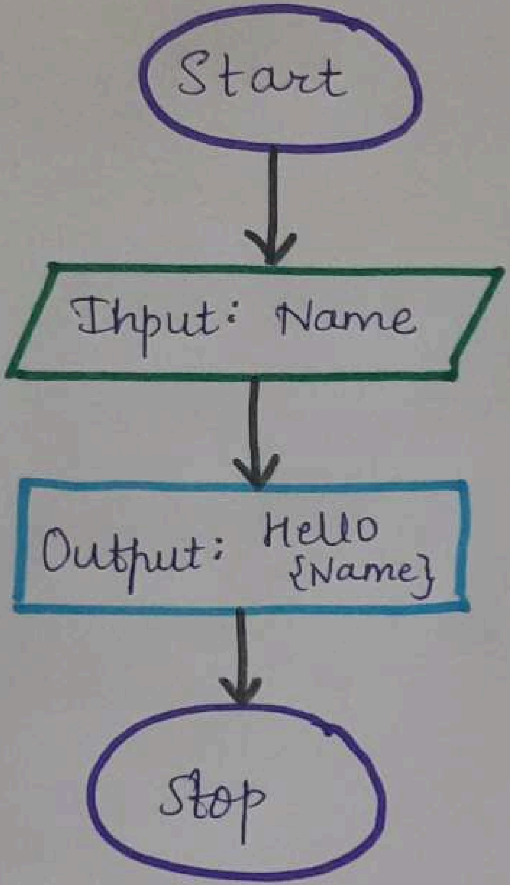


→ **Condition** → Represents for conditional statement



→ **Flow direction of program** → A line connector which shows what is the flow of program.

eg: Take a name & output Hello Name :



✱ Pseudo Code:

It is just a way to write steps which is human readable format. [It is not a code].

It is mainly meant for human reading not for machine reading.

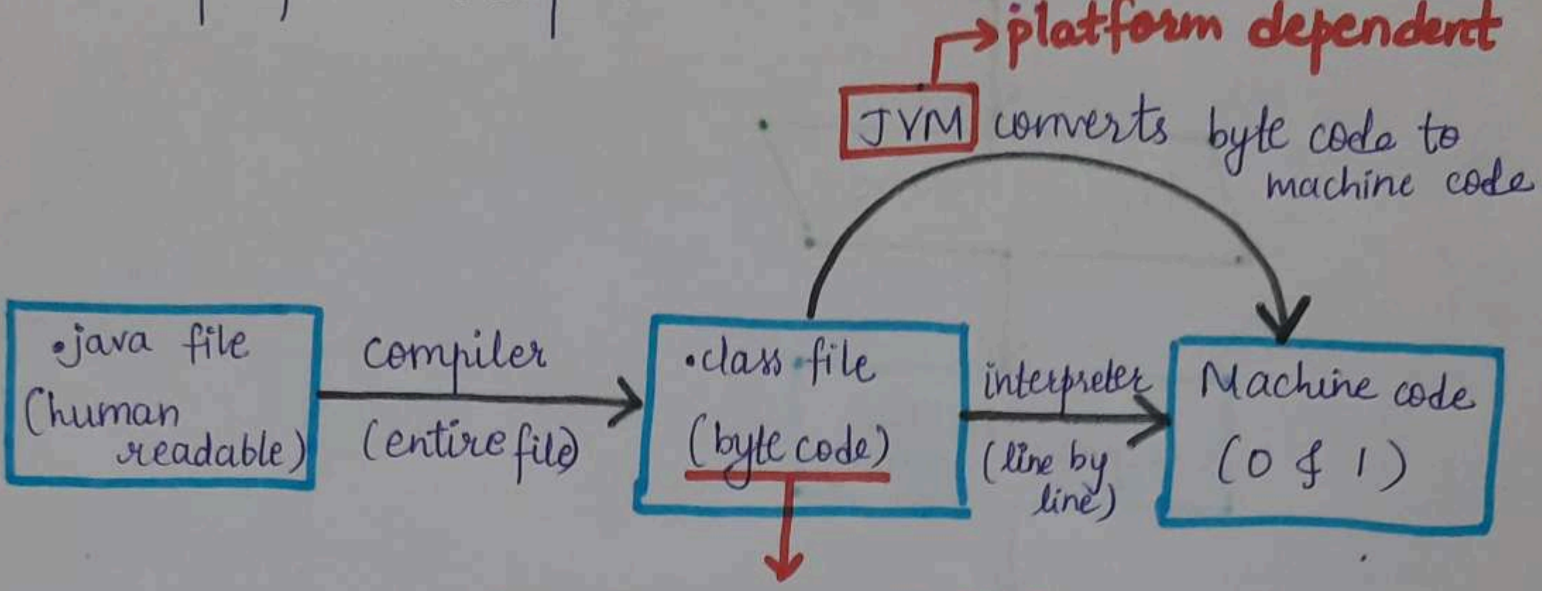
eg: Take above example to take a name & outpu Hello name :

```
Step 1 :→ Start
Step2 :→ Take input from user [name = Input ("enter na
Step3 :→ Hello {Name} [output]
Step4 :→ Stop
```

3/8/21

Introduction to Java

✱ How Java code executes and more information about platform Independence...



platform dependent
JVM converts byte code to machine code
.java file (human readable)
compiler (entire file)
.class file (byte code)
interpreter (line by line)
Machine code (0 & 1)

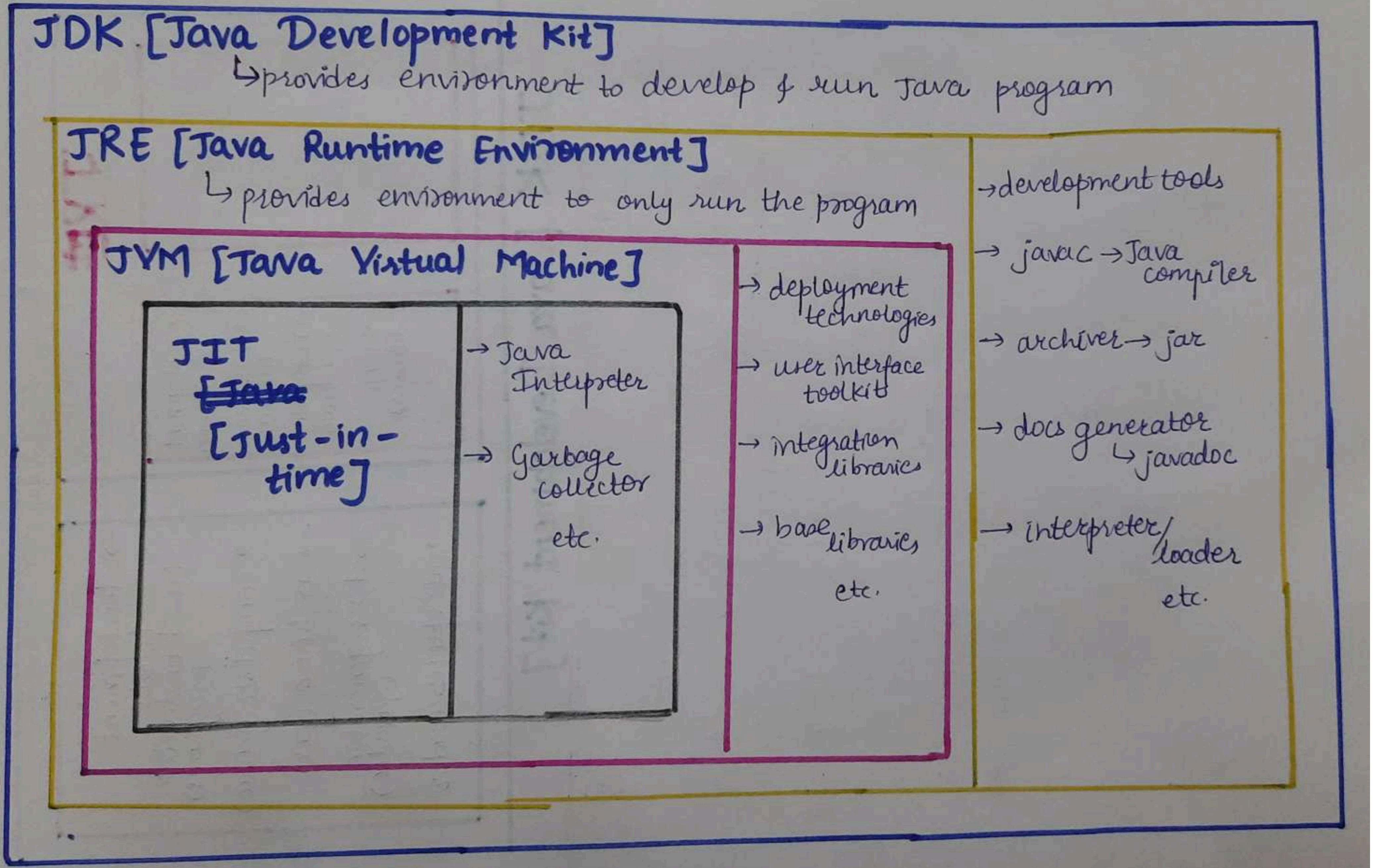
- **can run on all O.S.**
- **this code doesn't run directly on a system, for this we need JVM**

✱ Therefore, Java is platform independent ✱

⟹ We can provide this byte code to any system means we can compile the java code on any system.

⟹ But JVM is platform dependent means for every O.S. the executable file that we get, it has step by step set of instruction dependent on platform.

☆ JDK vs JRE vs JVM vs JIT



JDK [Java Development Kit]

↳ provides environment to develop & run Java program

JRE [Java Runtime Environment]

↳ provides environment to only run the program

JVM [Java Virtual Machine]

JIT ~~[Java~~ [Just-in-time]

→ Java Interpreter

→ Garbage collector

etc.

→ deployment technologies

→ user interface toolkit

→ integration libraries

→ base libraries

etc.

→ development tools

→ javac → Java compiler

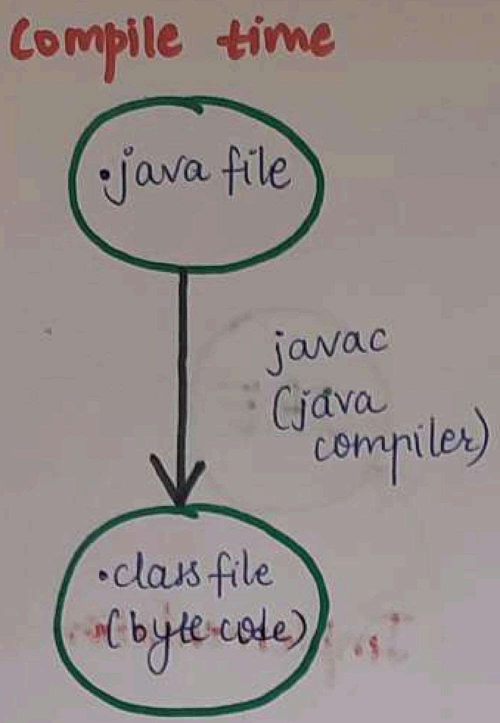
→ archiver → jar

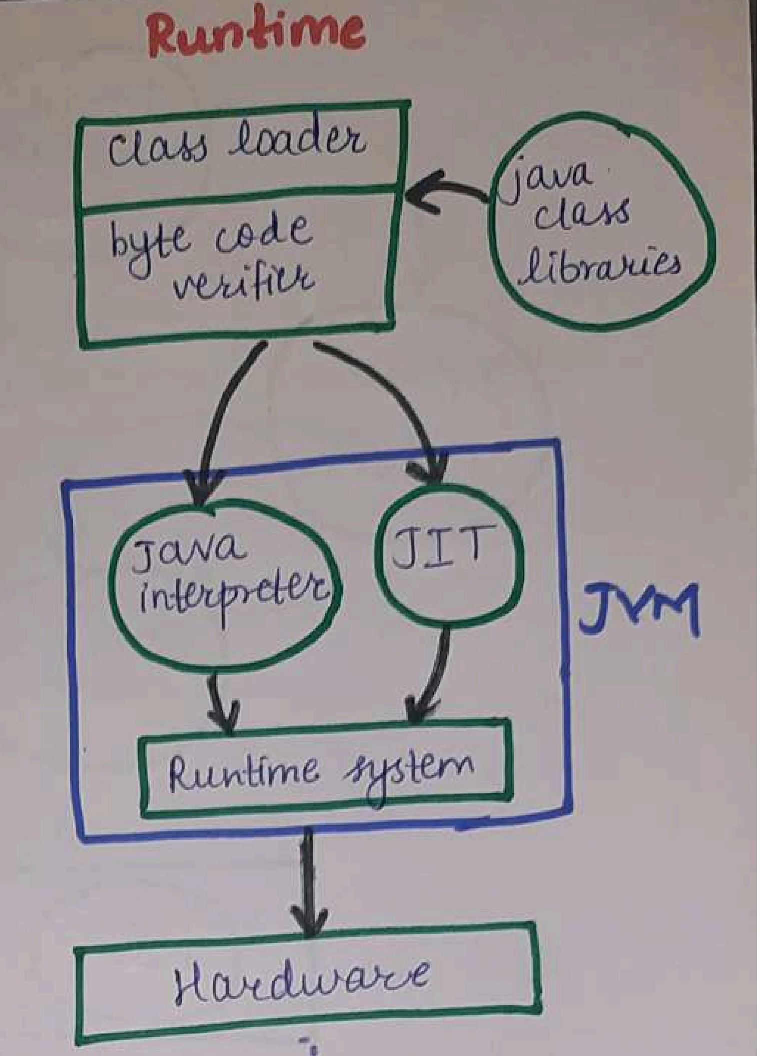
→ docs generator ↳ javadoc

→ interpreter/loader

etc.

✱ Java Development and Runtime Enviroment





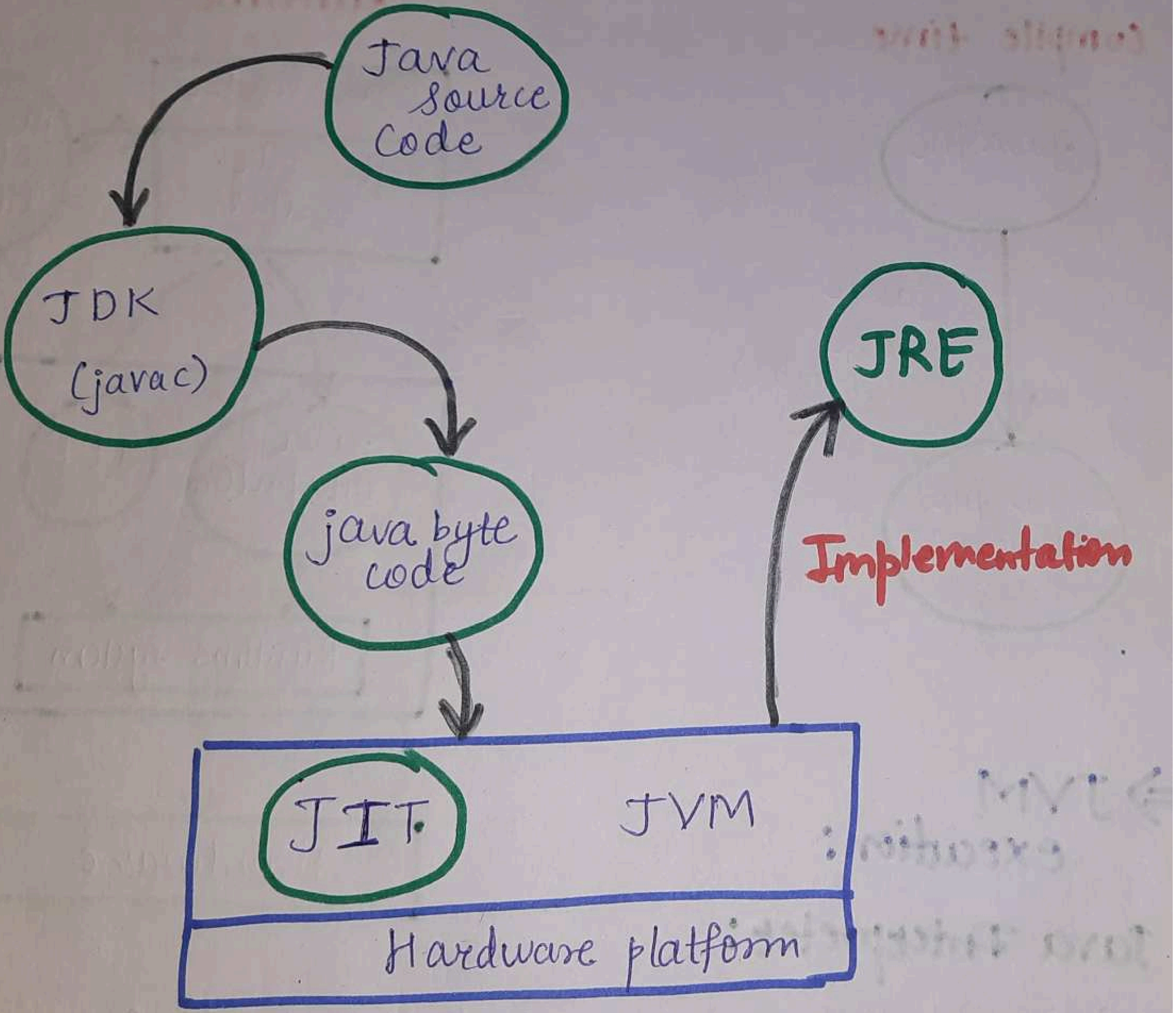
⇒ JVM execution:

- **Java Interpretet:**
  - → line by line execution
  - → when one method is called many times, it will interpret again & again
- **JIT:**
  - → methods that are repeated, JIT provides direct machine code so re-interpretation is not required
  - → makes execution faster
- **Garbage Collector**

✱ Class loader:

- **Loading**
  - → reads byte code file & generates binary data
  - → an object of this class is created in heap
- **Linking**
  - → JVM verifies .class file
  - → allocates memory for class variables & default values
  - → replace symbolic references from the type with direct references
- **Initialization**
  - → all static variable are assigned with their values defined in the code & static block

✯ Summary:


Java source code
JDK (javac)
java byte code
JRE
Implementation
JIT
JVM
Hardware platform

3|8|21

First Java Program - Input / Output, Debugging & Datatypes

File name: **Demo.java** Class Name: **Demo**

→ Its good practice to use initial character as capital (you can use small also)

public → this keyword means, it is used so that we can access the class from anywhere.

functions → Collection of code, that we can use again & again. Functions are also known as methods.

void → The void keyword specifies that a method should not have a return value.

String[] args → means an array of sequence of characters ("strings") that are passed to the main function.

[] → array

- After compiling, .class file is always saved in current location where you are in.
- If you want to change the location, use **-d** (destinatio option while compiling and specify the path.

```
javac -d <path> Demo.java
```

- **echo $PATH** → every command looks for this location before executing.

  [environment variables]

- class name & file name should be same, but if we don't want to make class name as file name then it should not be public.

for eg → **class Divide**

- package com.abc OR package com.defg

```
com
 ├─abc
 │  ├─file1
 │  └─file2
 └─defg
    ├─file1
    └─file2
```

- System.out.println("Hello"); → this means print the output on Standard output stream (here, terminal)

System → class

out → var

println → fn/method

println → adds new line

print → does not add new line.

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

Scanner → class that allows us to take input

new → creating object

System.in → take input from standard input (here, keyboard)

Primitive → means any data type that cannot be broke further.

integer, character etc. are primitive datatype.

⟹

```
int rollno = 64; → 4 bytes
char letter = 'r';
float marks = 98.67f; → 4bytes
double largeDecimalNumbers = 456789.12345; → 8bytes
long largeInteger = 123456781OL; → 8 bytes
boolean check = true;
```

- String is written in double quotes whereas while specifying char we write it in single quotes.
- All decimal values that we use are by default of double datatype, therefore if we want to store in float we have to use "f", same for int & long.

```
float marks = 7.2f
(by default) double largeDecimalNumbers = 456789101.12345

int rollno = 64; (by default)
long LargeInteger = 1234567891011L;
```

- Integer → Wrapper class → provides additional functionalities
  ↳ converts primitive datatype to object.

- Comment → the lines that we comment are ignored by Java and will not be executed.
  Comment in Java → //

→ int a = 10 → literal
  a ↓ identifier

- Literals : Java literals are syntactic representations of boolean, character, numeric or string data.
  here, 10 is an integer literal.

- Identifiers : Identifiers are the names of variables, methods, classes, packages & interfaces.

- int a = 234_000_000;

  ↳ the value of a will be 234000000, underscore will be ignored.

- 564.12345678 $\xrightarrow{\text{rounds off}}$ 564.12345

  If we give float very big, than it rounds off the value which gives floating point error.

⟹ Type Casting & Type Conversion :

- **Widening or Automatic Type Conversion :**

  → Two datatypes are automatically converted.

  → This happens when we assign value of smaller datatype to bigger datatype & two datatype must be compatible.

byte → short → int → long → float → double

```
eg →  int  i = 100;     →  100
      long  l = i;      →  100
      float  f = l;     →  100.0
```

- **Narrowing or Explicit Conversion:**

  → This happens we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

double → float → long → int → short → byte

```
eg →  double d = 100.04;    →  100.04
      long l = (long) d;    →  100
      int i = (int) l;      →  100
```

• Automatic Type Promotion in Expressions :

→ While evaluating expressions, the intermediate value may exceed the range of operands & hence the expression value will be promoted.

→ Some conditions of type promotion are:

1. Java automatically promotes each byte, short, char to int when evaluating an expression
2. Long, float or double the whole expression is promoted to long, ~~whole to~~ float or double.

eg: After solving expression :

$$(f * b) + (i / c) - (d * s);$$

we get →

$$\underbrace{\underline{\text{float}} + \underline{\text{int}} - \underline{\text{double}}}_{\text{converted to biggest one}} = \underline{\text{double}}.$$

• Explicit type casting in expressions :

→ If we want to store large value into small data type

eg :

```
byte b = 50;
b = (byte)(b*2); → type casting int to byte.
```

• If-else syntax in Java

```
if (condition) {
   // block of code
 } else {
    // block of code
}
```

• For loop syntax

```
for (statement1; statement2; statement3){
        // code block
}
```

6|8|21 If-else conditions

Loops → while & for & do-while

7|8|21 Switch Statements + Nested case in Java.

• Switch Statements:

```
switch (expression) {
        case one:
                // code block
                break;  → terminate the
                          sequence
        case two:
                // code block
                break;
        default:                  → default will
                //code block        execute when
                                    none of above
}                                   does.
                                  → if default is
                                    not at end put
                                    break after it.
```

→ if break is not used then it will continue with other cases.

→ duplicate cases not allowed.

eg:
```
case one:
        // code block
        break;
case one:
        //code block
        break;
```

✕ not allowed.

New syntax:

```
switch (expression) {
        case one →//do this ;
        case two → // do this;
        default → // do this;
    }
```

★ x. equals ("word") → here equals only checks value not reference.

x == "word" → here it checks reference

- **Nested Switch Case:**

```
switch (expression) {
        case one:
             // code block
             break;
        case two:
             switch (expression) {
                   case one :
                        //code block
                        break;
                   case two :
                        //code block
                        break;
                   default;
                        // code block
             }
             break;
    default: // code block
}
```

8/8/21 Functions/Methods in JAVA

Functions/Methods (in java):

- A method is a block of code which only runs when it is called.
- To reuse code: define the code once, & use it many times.

Syntax:

```
public class Main {
        static void myMethod() {
                //code
        }
}
```

void → this method myMethod() does not have a return value.

myMethod → name of method

```
public class Main {
        access-modifier  return_type  method() {
                //code
                return statement;
        }
}
```

return statement; → f^n ends here

- method() → calling the function.
  method ↓ name of function

- **return_type :-**

A return statement causes the program control to transfer back to the caller of a method.
A return type may be premitive type like int, char, or void type (returns nothing).

⇒ there are a few important things to understand about returning the values:

- The type of data returned by a method must be compatible with the return type specified by the method.

  eg: if return type of some method is boolean, we cannot return an integer.

- The variable recieving the value returned by a method must also be compatible with the return type specified for the method.

⇒ Pass by value:

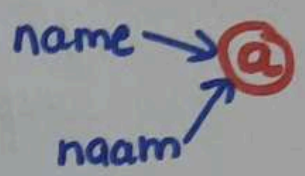
eg 1:

```
main() {
    name = "a";
    greet(name);
}
static void greet(naam) {
    print(naam)
}
...
```

object/value

Creating copy of value of name

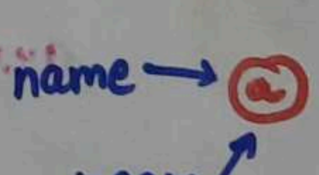
i.e., passing value of the reference.

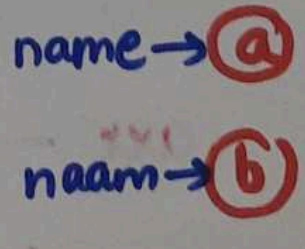


eg2:

```
psvm() {
    name = "a";
    change(name);
    print(name);
}
change(naam) {
    naam = "b";
}
```

Creating copy





not changing original object, just creating new object.

since it is created inside fn it will not change original one.

✲ points to be noted :

1→• primitive data type like int, short, char, byte etc.
↳ just pass value

2→• object & reference :
↳ passing value of reference variable.

eg-1 :

```
psvm () {
      a = 10;
      b = 20;
      swap(a, b);
}
swap (num1, num2) {
        temp = num1;
        num1 = num2;
        num2 = temp;
}
```

a → 10
b → 20
] but not here

temp → 10
num1 → 20
num2 → 10
] at fn scope level they are swapped.

Here, they just passes the value....

eg-2 :

arr → [1, 2, 3, 4, 5]
nums ↗

nums[0] = 99 [now, the value of 0th position in nums will change which also changes value of arr[0]]

arr → [99, 2, 3, 4, 5]
nums ↗

Here, passing value of reference variable

* Scopes:

- **function scope :**

variables declared inside a method / function scope (means inside method) can't be accessed outside the method.

eg:-

```
psvm(){
        ✗
}
all(){
  int x;
}
```

can't be accessed outside

- **block scope:**

```
psvm ( ) {
      int a = 10;
      int b = 20;
      {
       int a = 5; ✗
        a = 100; ✓
        int c = 20;
      }
      c = 10; ✗
      int c = 15; ✓
      a = 50; ✓
}
```

variables initialized outside the block can be updated inside the box.

variables initialized inside the block cannot be updated outside the box but can be reinitialized outside the block.

variables like "a" here, is declared outside the block, updated inside the block and can also be updated outside the block.

- **loop scope :**

variables declared inside loop are having loop scope.

⇒ Shadowing:

Shadowing in Java is the practice of using variables in overlapping scopes with the same name where the variable in low-level scope overrides the variable of high-level scope. Here the variable at high-level scope is shadowed by low-level scope variable.

eg:-
```
public class Shadowing {
        static int x = 90;
        psvm ( ) {.
                System.out.println (x);   → 90
                x = 50;                   // here high-level scope is shadowed by low-level scope
                System.out.println (x);   → 50
        }
}
```

⇒ Variable Arguments:

- Variable Arguments is used to take a variable number of arguments. A method that takes a variable number of arguments is a varargs method.

Syntax:

```
static void fun (int ...a) {
                // method body
        }
```

Here, parameters would be array of type int [ ] method)

⇒ Function Overloading:

Function Overloading happens when two functions have same name.

eg → 1)

```
fun ( ) {
    //code
}

fun ( ) {
    //code
}
```

× function overloading

2)

```
fun ( int a ) {
        //code
    }

fun ( int a, intb) {
        //code
    }
```

This is allowed having different arguments with same method name.

⇒ At compile time, it decides which f^n to run.

⇒ Armstrong number:

suppose there is number → 153

$$153 \longrightarrow (1)^3+(5)^3+(3)^3 = 1+125+27$$

$$= \underline{\underline{153}}$$

10/8/21. Introduction to Arrays & Arraylist in Java

Why do we need Arrays?

⇒ It was simple when we had to store just five integer numbers and now let's assume we have to store 5000 integer numbers. Is it possible to use 5000 variable? NO

To handle these situations, in almost all programming language we have a concept called **Array**.

Array is a data structure use to store a collection of data.

⇒ Syntax of an Array:

```
datatype[ ] variable_name = new datatype[size];
```

eg: we want to store roll numbers:

```
int[ ] rollnos = new int[5]
```

store 5 roll numbers

OR

```
int[ ] rollnos = {51, 82, 13, 15, 16}
```

represent the type of data stored in array.

All the type of data in array should be same!

⇒ Internal working of array:

```
int[ ] rollnos; //declaration of array
```

↳ rollnos are getting defined in stack

```
rollnos = new int[5]; //initialisation
```

↳ actual memory allocation happens here

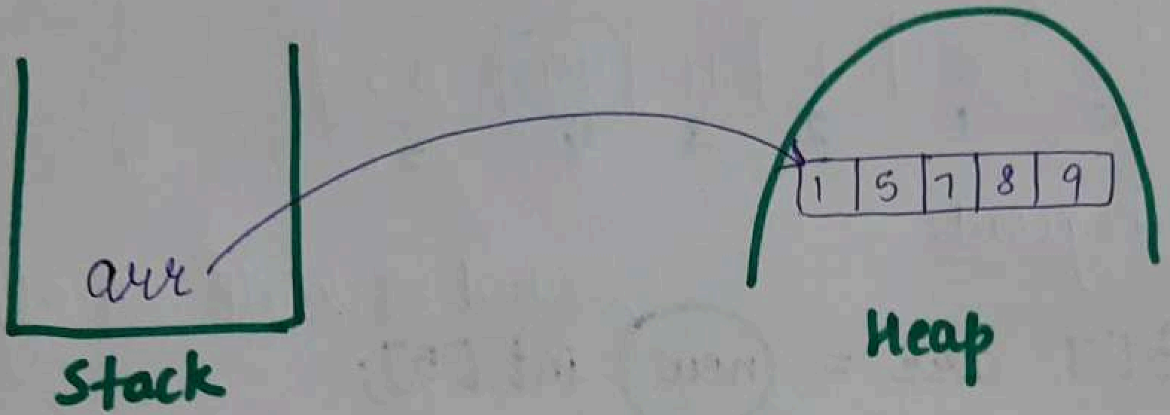
Here, object is being created in heap memory.

declaration of array — compile time   intialisation — run time

```
int [] arr = new int[5];
```

datatype. ref var   creating object in heap memory.

⇛ This above concept is known as <u>Dynamic memory allocation which means at runtime OR execution time memory is allocated.</u>



⇛ <u>Internal Representation of Array</u> :

- Internally in Java, memory allocation totally depends on JVM whether it be continuous or not !

<u>Reason 1</u> : Objects are stored in heap memory.

<u>Reason 2</u> : In JLS (Java Language Specification) it is mentioned that heap objects are not continuous

<u>Reason 3</u> : Dynamic memory allocation. Hence, array objects in Java may not be continous (depends on JVM)

⇒ Index of an array:

index →	0	1	2	3	4	5
arr	3	8	9	10	53	33

arr[0] = 3 arr[2] = 9 arr[4] = 53
arr[1] = 8 arr[3] = 10 arr[5] = 33

Suppose ~~to~~ change the value of certain index:

arr[4] = 99

New array will be:

3	8	9	10	99	33
0	1	2	3	4	5

⇒ new keyword:

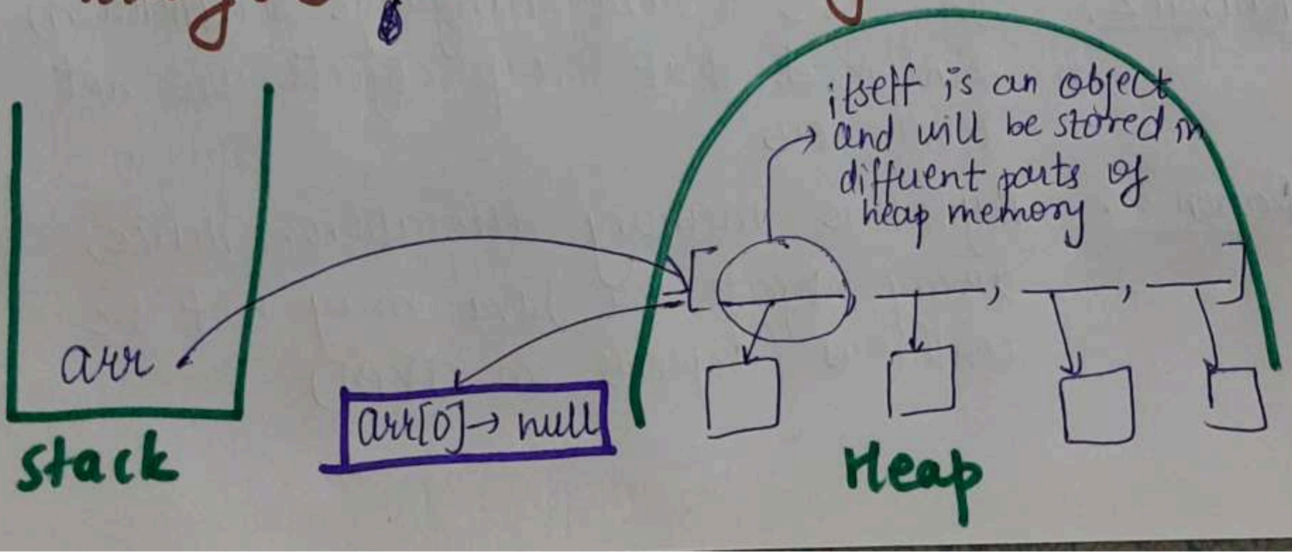
int[] arr = new int[5];

new → used to create an object

it will create an object in heap memory of array size 5.

⇒ If we dont provide values in the array, internally by default it stores [0, 0, 0, 0, 0]. for above size of array.

String[] arr = new String[4];


itself is an object and will be stored in diffuent parts of heap memory
arr
arr[0] → null
Stack
Heap

- ✱ primitives (int, char etc) are stored in stack.
- ✱ All other objects are stored in heap memory.

⟹ Arrays.toString (array) ⟶ internally uses for loop and gives the output in proper format.

- ✱ In an array, since we can change the objects, hence they are <u>mutable</u>.
- ✱ strings are immutable.

⟹ <u>2 D Array:</u>

$$\begin{array}{c|ccc} & & 3 & \\ \hline 3 & 1 & 2 & 3 \\ & 4 & 5 & 6 \\ & 7 & 8 & 9 \end{array}$$

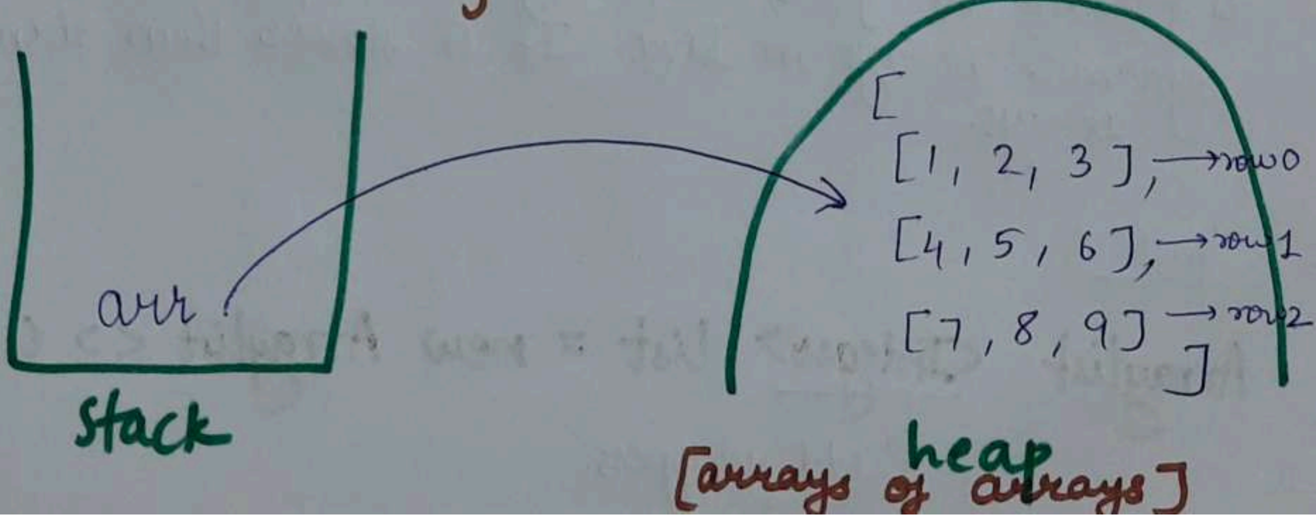
⟹ `int[][] arr = new int[size][ ]`

row ↓ (on [size]) — mandatory to give size of row

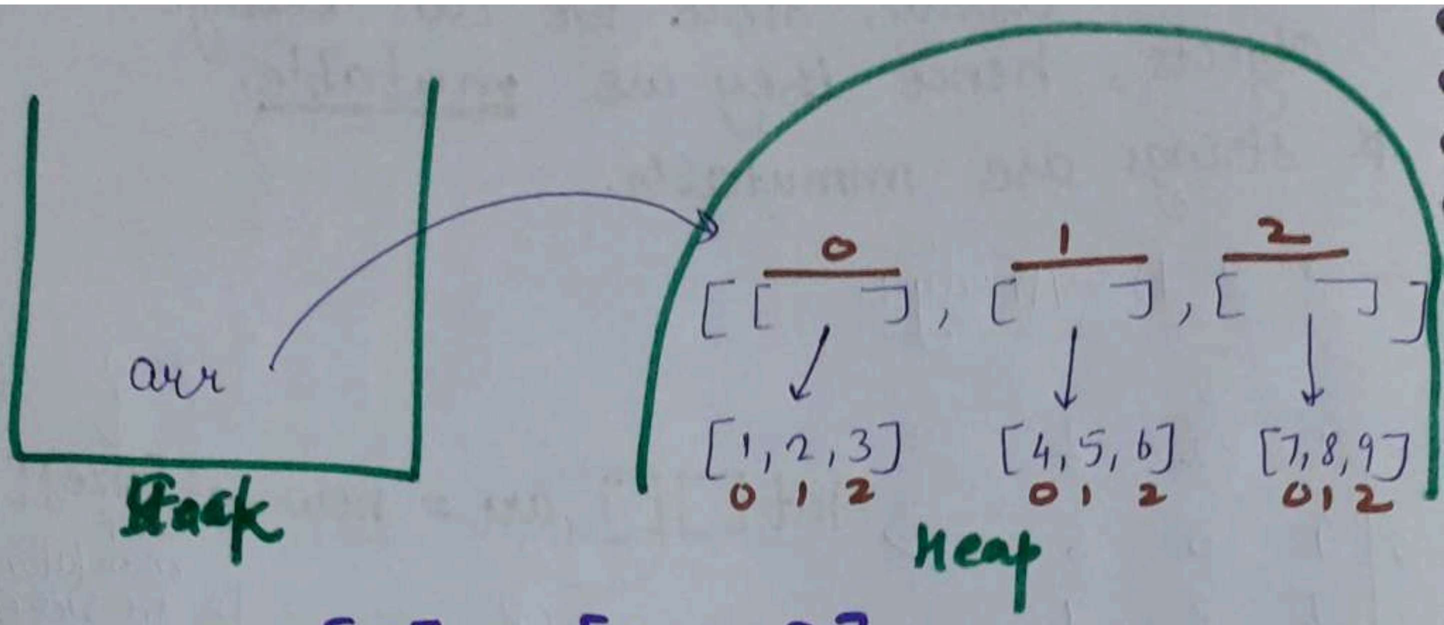
column ↓ (on [ ]) — not manda

OR

```
int[][] arr = {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 9}
              }
```



[arrays of arrays]



arr[0] = [1, 2, 3]
arr[0][2] = 3

⇒ Arraylists:

Arraylist is a part of collection framework and is present in java.util.package. It provides us with dynamic arrays in Java. It is slower than standard arrays.

Syntax :

```
Arraylist <Integer> list = new Arraylist <> ( );
```

↳ add wrappers.

⇒ Internal Working of Arraylist :

- size is fixed internally
- Suppose arraylist gets filled by some amount
  a) It will make an arraylist of say double the size of arraylist initially.
  b) Old elements are copied in the new arraylist.
  c) Old ones are deleted.

Strings and StringBuilder in Java

* What is String?

String is basically a collection/sequence of characters. and it is stored in String data type.

Example

$$\underbrace{\text{String}}_{\text{String declaration}}\ \underbrace{\text{name}}_{\text{reference variable}} = \underbrace{\text{"Kunal Kushwaha"}}_{\text{value}}$$

datatype

object (collection of character)

⇒ String is the most commonly used class in the Java's class library.

String name = "Kunal *Kushwaha"

Everything that start with capital letter is a class.

⇒ Every String that we create, it's actually an object of type String.

* Internal Working of String :—

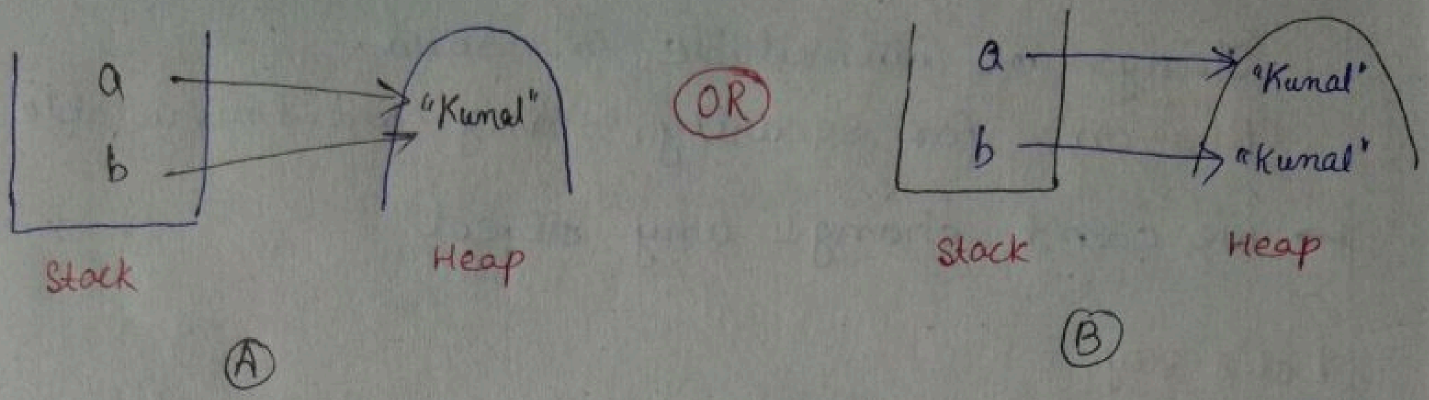
Let say,

String a = "Kunal"

String b = "Kunal"

Q. Is this creating two different objects or is it pointing to same object?

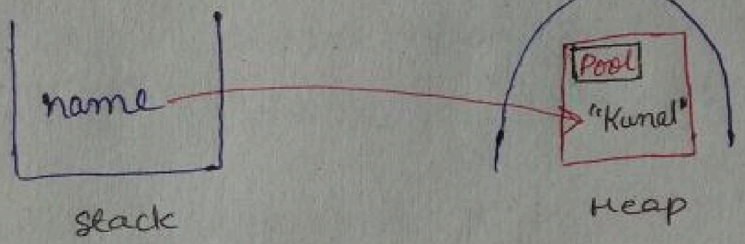
How it is stored (A) or (B) ?



Regarding this let's understand some concepts:-

1. String Pool :- It is a separate memory structure inside the heap.

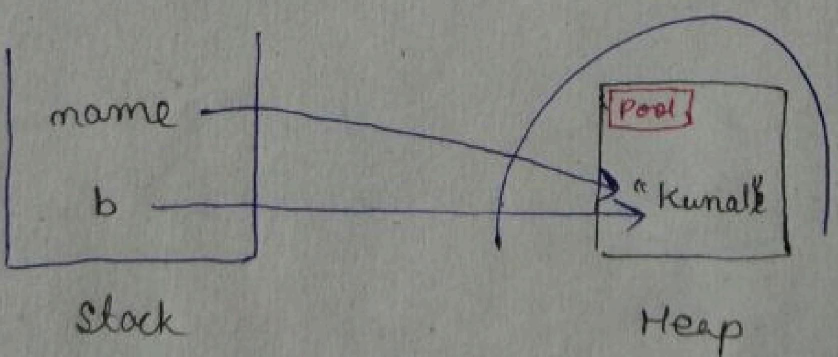
Ex- String name = "Kunal"



- Use of Pool :-

→ All the simillar values of strings are not recreated in the pool. That makes our programs more optimized.

Ex→ String name = "Kunal" ; String b = "Kunal"



Here, it says that "kunal" already exists in the pool. So, no need to create it again. Hence, point b to 'Kunal'.

2. Immutability :-

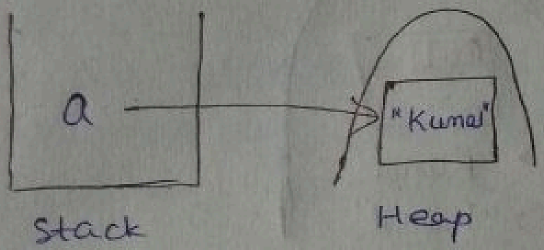
→ Strings are immutable in Java.

Reason :- For Security

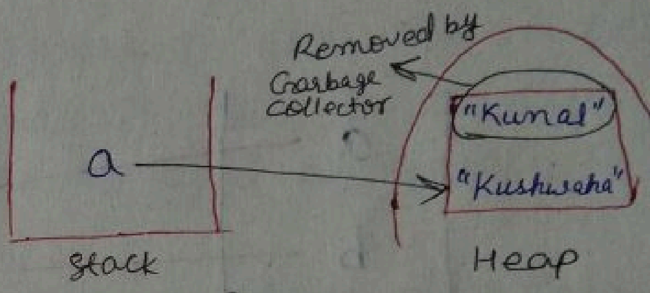
→ we can't change any object.

* Let's say :—

Initially ;- String a = "Kunal"



Then, a = "Kushwaha"



Here, we haven't change the object i.e "Kunal". we have created a new object i.e. "Kushwaha"

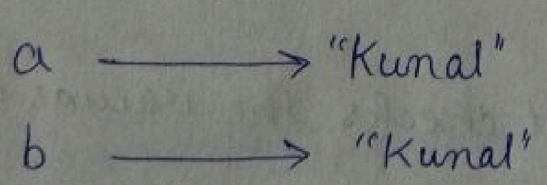
* String Comparison Methods :-

① == method :-

== ⇒ a comparator

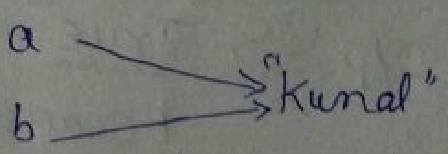
It checks ~~the~~ if the reference variables are pointing to same object

case - A



⇒ a==b will give False

case - B



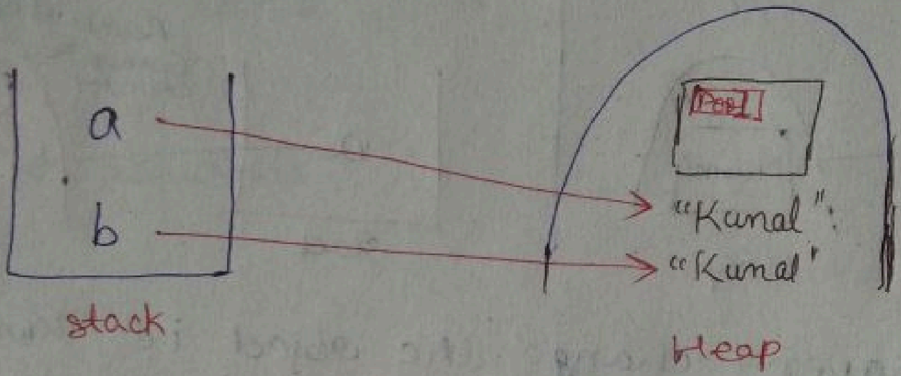
⇒ a==b will give True

* How to create different objects of same value :-

⇒ For this, we use "new" keyword.

```
String a = new String ("Kunal")
String b = new String ("Kunal")
```

// creating these values outside the pool but in heap.



NOTE :- This time a==b will give False.

② **.equals method** :- when we only need to check value, we use .equals method.

Ex

```
String a = new String ("Kunal");
String b = new String ("Kunal");
System.out.println (a.equals(b));
```

O/P ⇒ True

Because, it just checks the values are same or not.

* PrintStream — a class in java.

* out — a variable of type PrintStream.

* println — a method of PrintStream class.

NOTE:
Internal working of println → println calling the valueOf ~~function~~ method and that is calling toString() and then it's returning the string.

* Pretty Printing :— It prints/present the source code in an attractive way, so that it can be easily analyzed by the interpreter as well as easily read by humans.

Ex → Print the value of π till 3 digit after decimal.

```
System.out.printf("Pie: %.3f", Math.PI);
```

Print formatted string

placeholder

* System.out.println('a' + 'b');
O/P ⇒ 195
[ASCII value of a = 97,
ASCII value of b = 98]

* System.out.println("a" + "b");
O/P ⇒ ab   // String concatenation

* System.out.println('a' + 3);
O/P ⇒ 100   [ASCII value of a = 97]

* System.out.println((char)('a' + 3));
O/P ⇒ d

* System.out.println("a" + 1); // String "a" is not converting into its ASCII value .....

O/P ⇒ a1

NOTE : when an integer is added with a string it is converted to its rapper class integer. i.e, it is going to use tostring().

** String Performance ** [V.V.I]

Ex

```
public static void main(String[] args) {
            String series = " ";
            for(int i=0;  i<26 ;  i++) {
                  char ch = (char)('a'+ i);
                  series  = series + ch;
            }
            System.out.println(series);
}
```

O/P ⇒ abcdefghijklmnopqrstuvwxyz

Let's see the working of above code, And what is the problem? why it is not a very good solution?

⇒ Initially, series = " " // empty string

⇒ After 1st iteration ⇒ series = " " + 'a' = "a"

⇒ After 2nd iteration ⇒ series = "a" + 'b' = "ab"

⇒ After 3rd iter. ⇒ Series = "ab" + 'c' = "abc"

** Explaination **

⇒ So, we noticed that, new object is created everytime It is not changing the original object as we know that strings are immutable.
So, it's actually creating new string object and copying the old one and then appending the new changes..

⇒ That's why, there is so much wastage of memory becoz, all the objects are dereferenced.
It happens like ↓

a, ab, abc, abcd, abcde, abcdef, -----------
------ abcdefghijklmnopqrstuvwxy

All these above large strings will have no reference variable. i.e, wastage of memory.

⇒ These are of size ↓

$$1+2+3+4+5+6+ \cdots + N$$

$$= \frac{N(N+1)}{2} = O\left(\frac{N^2\cancel{+N}}{\cancel{2}}\right) = O(N^2)$$

* Solution →

StringBuilder :- It is a class just like string.
⇒ A datatype that allow us to modify the value.
⇒ It will not create a new object like string. but actually add in the original one.
i.e, StringBuilder is mutable.

```
public static void main(String[] args) {
        StringBuilder builder = new StringBuilder();
        for (int i = 0;  i < 26;  i++) {
              char ch = (char) ('a' + i);
              builder.append(ch);
        }
        System.out.println(builder.toString());
}
```

NOTE :- It gives O(N) complexity.

* String Methods :-

* toCharArray() :→ It converts the String into character array.

* length() :→ gives the length of String

* getBytes()

* toLowerCase() :→ prints the string into lowercase.

There are many more such methods ......