GitHub
GIT CHEAT SHEET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history
`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker
`git config --global color.ui auto`
set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository
`git clone [url]`
retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit
`git add [file]`
add a file as it looks now to your next commit (stage)
`git reset [file]`
unstage a file while retaining the changes in working directory
`git diff`
diff of what is changed but not staged
`git diff --staged`
diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`
list your branches. a * will appear next to the currently active branch
`git branch [branch-name]`
create a new branch at the current commit
`git checkout`
switch to another branch and check it out into your working directory
`git merge [branch]`
merge the specified branch's history into the current one
`git log`
show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`	show the commit history for the currently active branch
`git log branchB..branchA`	show the commits on branchA that are not on branchB
`git log --follow [file]`	show the commits that changed file, even across renames
`git diff branchB...branchA`	show the diff of what is in branchA that is not in branchB
`git show [SHA]`	show any object in Git in human-readable format

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`	add a git URL as an alias
`git fetch [alias]`	fetch down all the branches from that Git remote
`git merge [alias]/[branch]`	merge a remote branch into your current branch to bring it up to date
`git push [alias] [branch]`	Transmit local branch commits to the remote repository branch
`git pull`	fetch and merge any commits from the tracking remote branch

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`	delete the file from project and stage the removal for commit
`git mv [existing-path] [new-path]`	change an existing file path and stage the move
`git log --stat -M`	show all commit logs with indication of any paths that moved

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`	apply any commits of current branch ahead of specified one
`git reset --hard [commit]`	clear staging area, rewrite working tree from specified commit

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`	system wide ignore pattern for all local repositories

TEMPORARY COMMITS

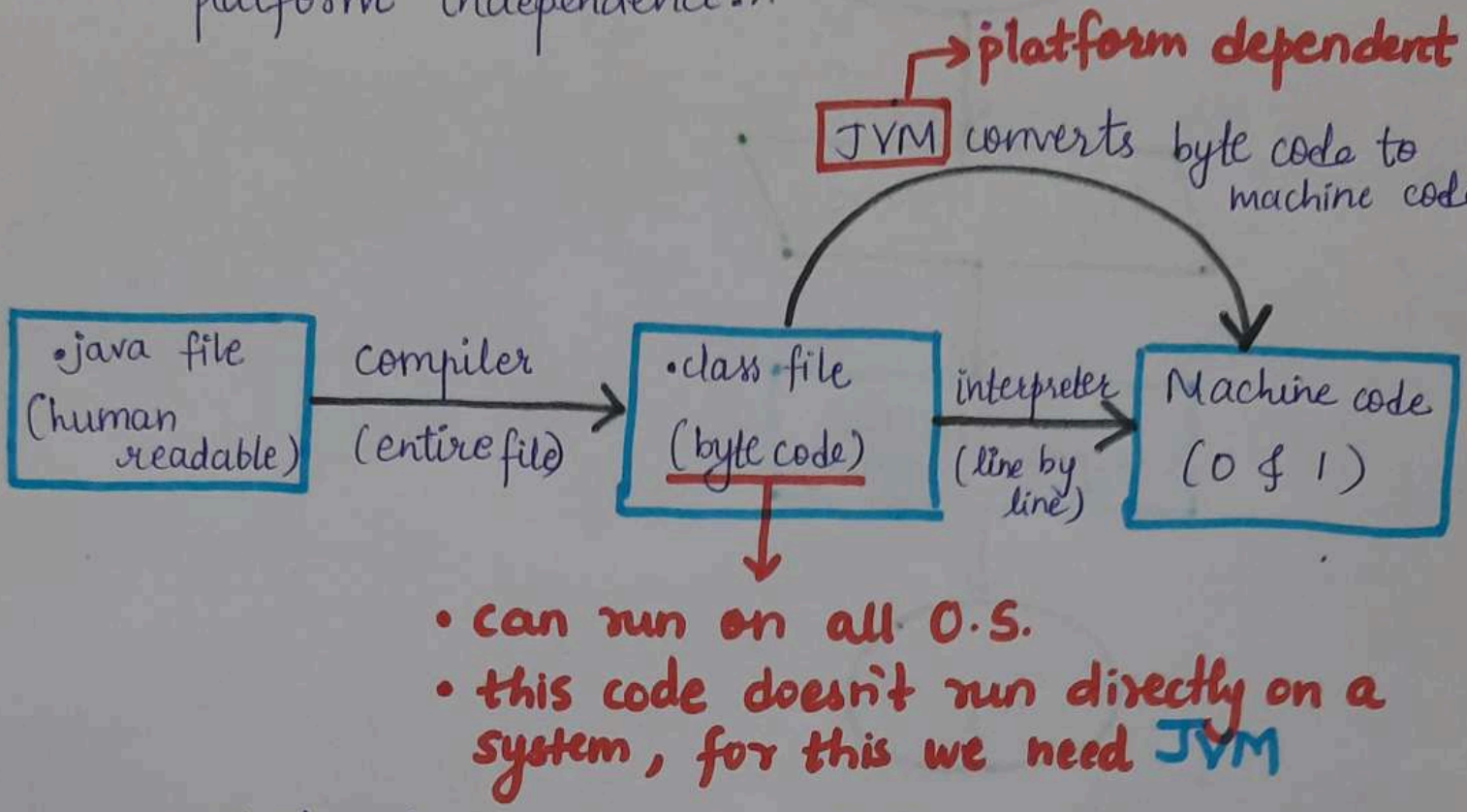
Temporarily store modified, tracked files in order to change branches

`git stash`	Save modified and staged changes
`git stash list`	list stack-order of stashed file changes
`git stash pop`	write working from top of stash stack
`git stash drop`	discard the changes from top of stash stack

3/8/21

Introduction to Java ♥

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



- 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

- → deployment technologies
- → user interface toolkit
- → integration libraries
- → base libraries

etc.

JVM [Java Virtual Machine]

JIT [Just-in-time]

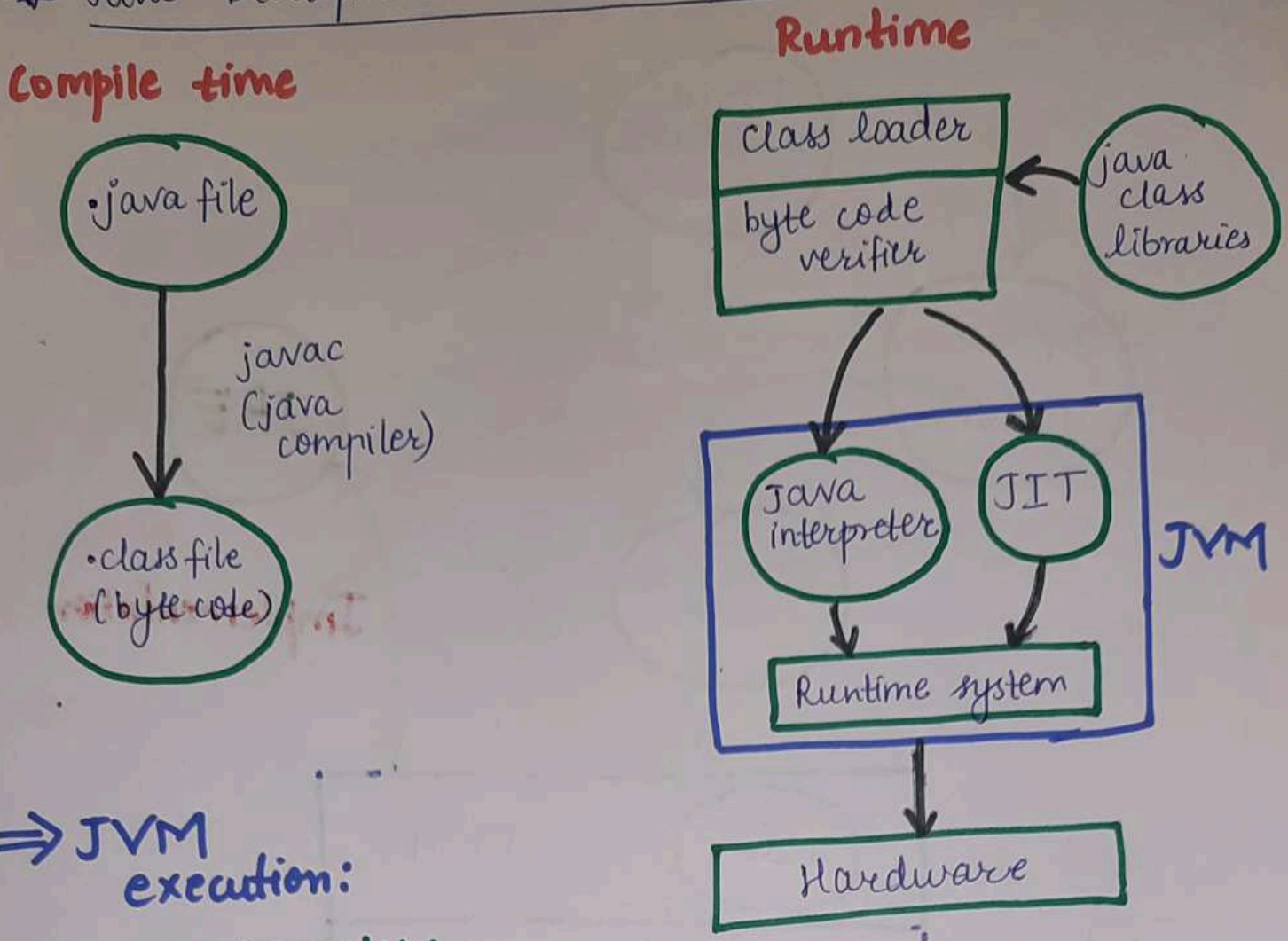
- → Java Interpreter
- → Garbage collector

etc.

- → development tools
- → javac → Java compiler
- → archiver → jar
- → docs generator ↳ javadoc
- → interpreter/loader

etc.

✱ Java Development and Runtime Environment



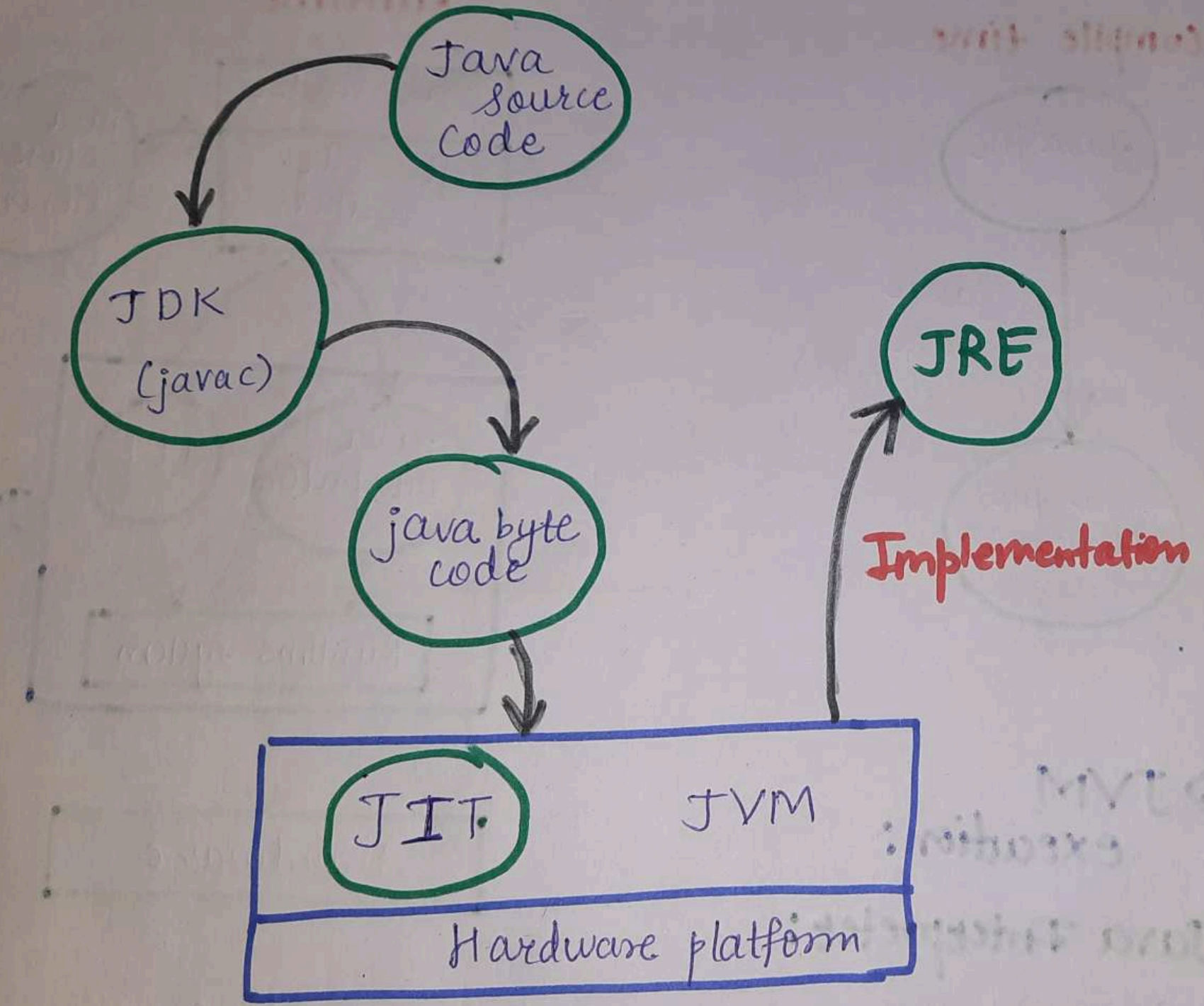
⇒ JVM execution:

- **Java Interpreter:**
  - → 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.

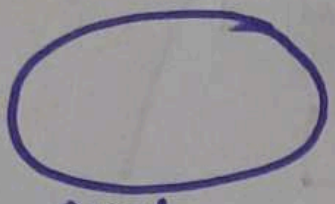
Summary:


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

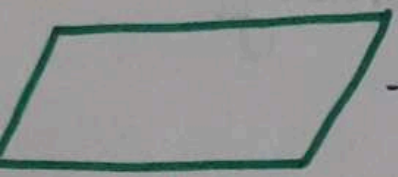
2/8/21

Flow of program

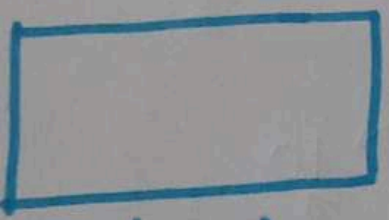
* Flow Chart Symbols :



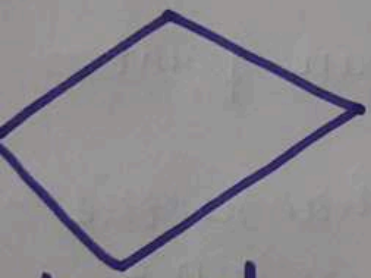
oval → Start/Stop → Represents start or end point of program.



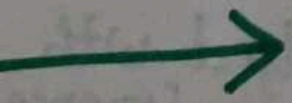
parallelogram → Input/Output → Represents the input & output



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

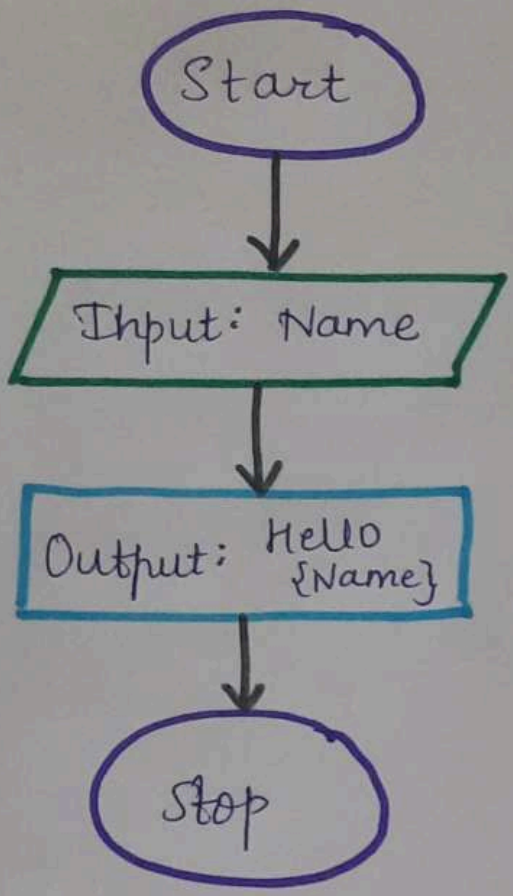


diamond → Condition → Represents for conditional statement



→ 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 & output Hello name :

Step 1 : → Start

Step2 : → Take Input from user [name = Input ("enter na

Step3 : → Hello {Name} [output]

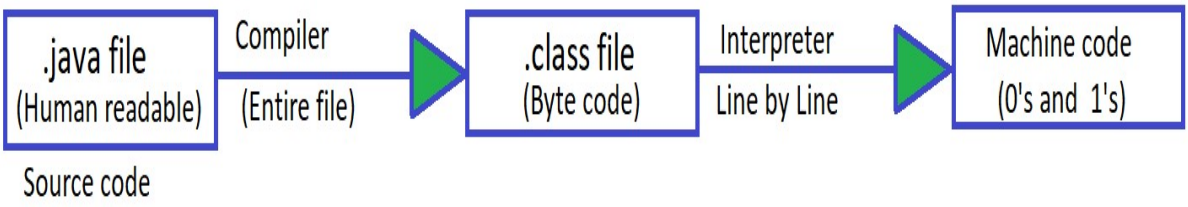
Step4 : → Stop

INTRODUCTION TO JAVA

Ques – Why do we use Programming language?

Ans – Machine only understand 0's and 1's, for humans it is very difficult to instruct computer in 0's and 1's so to avoid this issue we write our code in human readable language (Programming language).

" Java is one of the Programming Language"



- The code written in java is human readable and it is saved using extension **.java**
- This code is known as source code

Java Compiler:-

- Java compiler converts the source code into byte code which have the extension **.class**
- This byte code not directly run on system
- We need JVM to run this
- Reason why java is platform independent

Java Interpreter

- Converts byte code to machine code i.e. 0's and 1's
- It translate the byte code line by line to machine code

More About Platform Independent

→ It means that byte code can run on all operating system

→ We need to convert source code to machine code so computer can understand it.

→ Compiler helps in doing this by turning it into executable code.

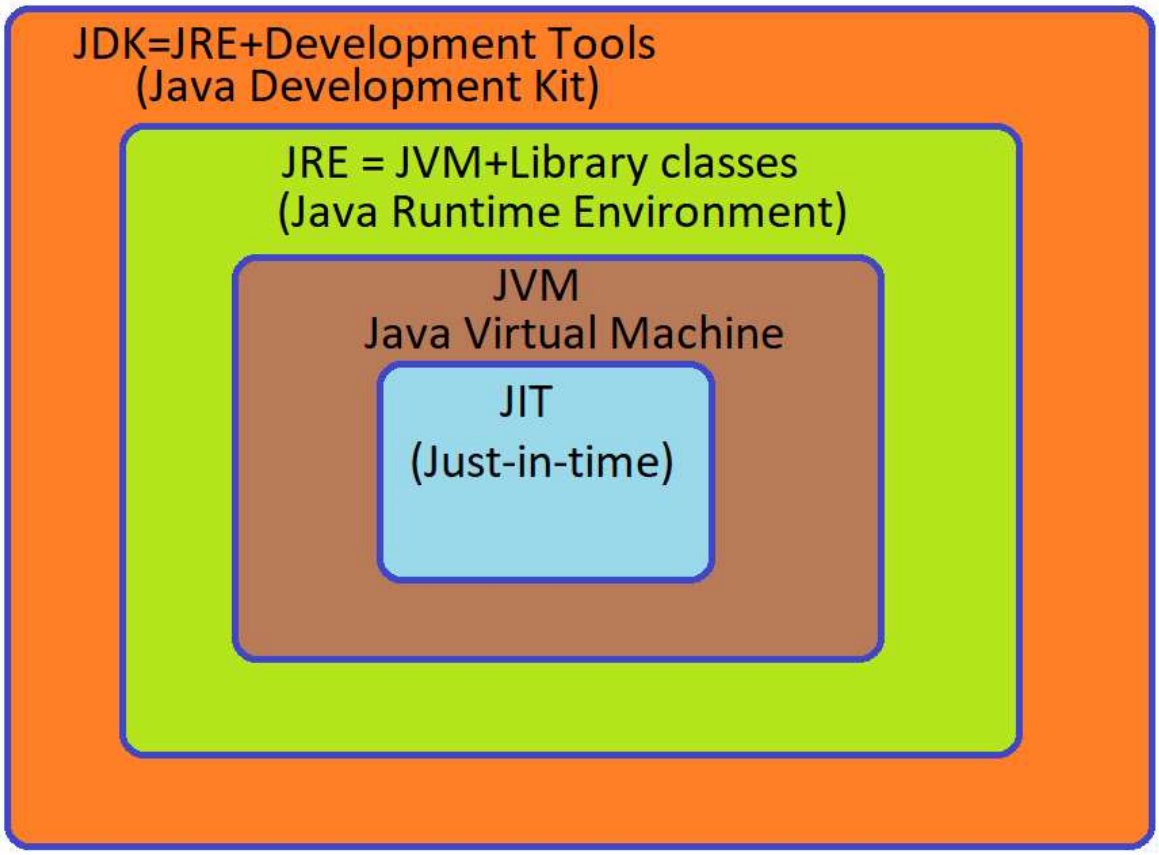
→ This executable code is a set of instruction s for the computer

→ After compiling C/C++ code we get .exe file which is platform dependent.

→ In java we get byte code, JVM converts this to machine code.

→ Java is platform independent but JVM is platform dependent.

Architecture of Java



JDK

- Provide Environment to develop and run the java program.
- It is a package that includes :-
  1. **Development tools :-** To provide an environment to run your program.
  2. **JRE :-** To Execute your program.
  3. **A compiler :-** javac
  4. **Archiver :-** Jar
  5. **Docs generator :-** Javadoc
  6. **Interpreter/loader**

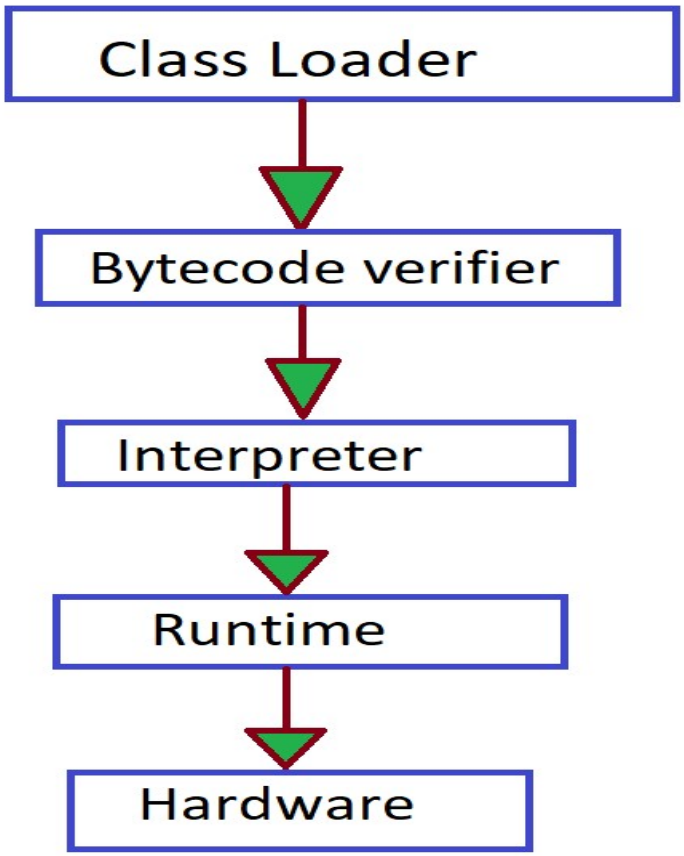
JRE

- It is an installation package that provides environment to only run the program.
- It consist of :-
  1. **Deployment technology**
  2. **User interface toolkit**
  3. **Integration libraries**
  4. **Base libraries**
  5. **JVM** :- Java virtual Machine

Compile Time:-

- After we get the .class file the next thing happen at runtime :
  1. Class loader loads all classes needed to execute the program.
  2. JVM sends code to bytecode verifier to check the format of code.

Runtime :-



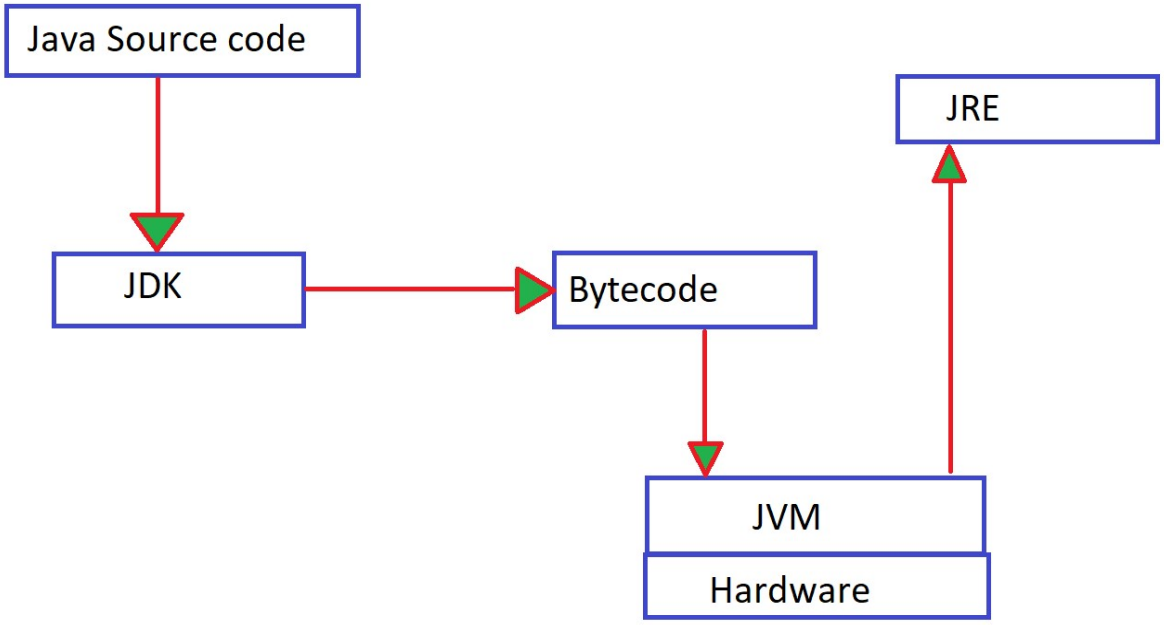
(How JVM Works) class Loader

- ***Loading***
  → Read .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 and default values
  → replace symbolic references from the type with direct reference.
- ***Initialization***
  → All static variables are assigned with their values defined in the code and static bock.
  → JVM contains the stack and heap memory locations.

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 that interpretation is not required.
  → Makes execution Faster.
  → **Garbage collector**

Working of Java Architecture



Tools required to run java on machine

1. **JDK** https://www.oracle.com/in/java/technologies/javase-downloads.html

2. **InteliJ**
   a) For windows :- https://www.jetbrains.com/idea/download/#section=windows
   b) For macOS :- https://www.jetbrains.com/idea/download/#section=mac
   c) For Linux :- https://www.jetbrains.com/idea/download/#section=linux

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** (destinato option while compiling and specify the path.

javac -d <path> Demo.java

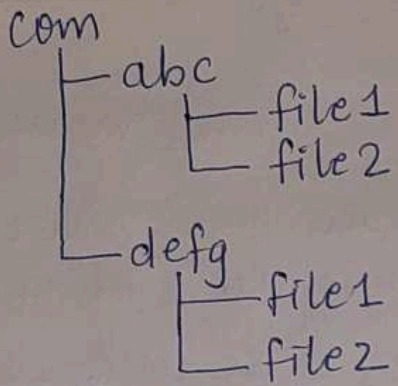
- **echo $PATH** → every command looks for this location before executing.

[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



- `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;  → 4 bytes
double largeDecimalNumbers = 456789.12345;  → 8 bytes
long largeIntegers = 12345678l0L;  → 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~~ float or double.

eg: After solving expression :

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

we get → float + int − double = double

converted to biggest one

• 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
}
```

CONDITIONAL AND LOOPS

Condition:- It provide check for the statement.

1. If-else statement → Used to check the condition, it checks the Boolean condition True or False.
   Syntax :-

```
if (boolean expression True or false){
    //Body
} else{
    // Do this
}
```

Example:-

```
public class IfElse {
    public static void main(String[] args) {
        int salary = 25400;
        if (salary> 10000) {
            salary = salary + 2000;
        }
        else {
            salary = salary + 1000;
        }

        System.out.println(salary);

    }
}
```

Output :- 27400

2. Multiple if-else statement

→ It executes one condition from multiple statements.

Syntax :-

```
if (condition 1){
    // code to be executed if condition 1 is true
} else if (condition 2) {
    // code to be executed if condition 2 is true
} else if (condition 3){
    // code to be executed if condition 3 is true
} else {
    // code to be executed if all conditions are false
}
```

Example :-

```
public class MultipleIfElse {
    public static void main(String[] args) {
        int salary = 25400;
        if (salary<= 10000) {
            salary +=1000;
        }
        else if (salary <= 20000) {
            salary += 2000;
        }
        else {
            salary += 3000;
        }
        System.out.println(salary);

    }
}
```

Output :- 28400

Loop → Loops are used to iterate a part of program several times.

1. for loop :- It is generally used when we know how many times loop will iterate.

   Syntax :-

```
for (initialization; condition; increment/decrement){
    // body
}
```

Example:- print numbers from 1 to 5

```
public class forloop {
    public static void main(String[] args) {
        for (int num=1;num<=5;num+=1){
            System.out.println(num);
        }
    }
}
```

Output :- 1
2
3
4
5

Example 2 :- print numbers from 1 to n

```
import java.util.Scanner;
public class forloop {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int num=1;num<=n;num+=1){
            System.out.print(num + " ");
        }
    }
}
```

Input : 6

Output :- 1 2 3 4 5 6

2. While Loop :- It is used when we don't know how many time the loop will iterate.

   Syntax :-

```
while (condition){
    // code to be executed
    // increment/decrement
}
```

   Example :-

```
public class whileloop {
     public static void main(String[] args) {
         int num = 1;
         while (num <=5){
             System.out.println(num);
             num += 1;
         }
     }
}
```

   Output :- 1
   2
   3
   4
   5

3. do while loop :- It is used when we want to execute our statement at least one time.
   → It is called exit control loop because it checks the condition after execution of statement.
   Syntax :-

```
do{
    // code to be executed
    // update statement -> increment/decrement
}while (condition);
```

Example :-

```
public class doWhileloop {
    public static void main(String[] args) {
       int n = 1;
       do{
           System.out.println(n);
           n++;
       } while(n<=5);
    }
}
```

Output :- 1
2
3
4
5

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)

■ Program to find largest of three numbers.

"Take 3 integer input from keyboard, Find the largest numbers among them ".

Approach -1 :-

```
import java.util.Scanner;
public class LrgestOfThree {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();

        int max = a;
        if(b>max){
            max = b;
        }
        if (c > max){
            max = c;
        }
        System.out.println(max);
    }
}
```

Approach – 2:-

```
import java.util.Scanner;
public class LrgestOfThree {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();

        int max = 0;
        if(a > b){
            max = a;
        } else {
```

```
            max = b;
        }
        if (c > max){
            max = c;
        }
        System.out.println(max);
    }
}
```

Approach 3 :-

Using Math.max :- Math is a class present in java.lang package and max is a function present in it which takes two number as an argument and return maximum out of them.

```
import java.util.Scanner;
public class LrgestOfThree {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int a = in.nextInt();
        int b = in.nextInt();
        int c = in.nextInt();

        int max = Math.max(c,Math.max(a,b));
        System.out.println(max);
    }
}
```

Input :- 3 6 5

Output :- 6

■ Alphabet case check

" Take an input character from keyboard and check weather it is Upper case alphabet or lower case alphabet"

```
import java.util.Scanner;
public class AlphabetCaseCheck {
    public static void main(String[] args) {
        Scanner in = new Scanner (System.in);
        char ch = in.next().trim().charAt(0);
        if (ch > 'a' && ch <= 'z'){
            System.out.println("Lowercase");
        }
        else {
            System.out.println("Uppercase");
        }
    }
}
```

Input :- a

Output :- Lowercase

Input :- Z

Output :- Uppercase

■ Fibonacci Numbers :- a series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers.

Ex :- 0,1,1,2,3,5,8,13...

→ **Find the nth Fibonacci number.**

" Given three input a, b, n a is the starting number of Fibonacci series and b is the next number after a, n is an number to find the nth Fibonacci number"

```
import java.util.Scanner;
public class FibonacciNumbers{
    public static void main(String[] args) {
        Scanner in = new Scanner (System.in);
        int n = in.nextInt();
        int a = in.nextInt();
        int b = in.nextInt();
        int count = 2;

        while(count <= n){
            int temp = b;
            b = b+a;
            a = temp;
            count++;
        }
        System.out.println(b);
    }
}
```

Input :- 0 1 7

Output :- 8.

- **Counting occurrence :-**

  " Input two numbers, find that hoe many times second number digit is present in first number"

  Ex :- first number = 14458

  Second number = 4

  Output = 2, because 4 is present 2 times in first number.

```
import java.util.Scanner;

public class CountingOccurence {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int count = 0;
        int Fn = in.nextInt();
        int Sn = in.nextInt();
        while (Fn>0){
            int rem = Fn % 10;
            if (rem == Sn){
                count++;
            }
            Fn = Fn/10;
        }
        System.out.println(count);
    }
}
```

Input :- 45535 5

Output :- 3

■ Reverse a number

" A number I input from the keyboard and Show the output as Reverse of that number "

Example :- Input :- 12345

Output :- 54321

```java
import java.util.Scanner;
public class ReverseANumber {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int num = in.nextInt();
        int ans = 0;
        while(num > 0){
            int rem = num % 10;
            num /= 10;
            ans = ans * 10 + rem;
        }
        System.out.println(ans);
    }
}
```

Input :- 458792

Output :- 297854

Calculator Program

```
import java.util.Scanner;

public class Calculator {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        // Take input from user till user does not press X or x
        int ans = 0;
        while (true) {
            // take the operator as input
            System.out.print("Enter the operator: ");
            char op = in.next().trim().charAt(0);

            if (op == '+' || op == '-
' || op == '*' || op == '/' || op == '%') {
                // input two numbers
                System.out.print("Enter two numbers: ");
                int num1 = in.nextInt();
                int num2 = in.nextInt();

                if (op == '+') {
                    ans = num1 + num2;
                }
                if (op == '-') {
                    ans = num1 - num2;
                }
                if (op == '*') {
                    ans = num1 * num2;
                }
                if (op == '/') {
                    if (num2 != 0) {
                        ans = num1 / num2;
                    }
                }
                if (op == '%') {
                    ans = num1 % num2;
                }
            } else if (op == 'x' || op == 'X') {
                break;
            } else {
                System.out.println("Invalid operation!!");
            }
            System.out.println(ans);
        }
    }
}
```

Input and output:-

```
Enter the operator: +
Enter two numbers: 86  94
180
Enter the operator: -
Enter two numbers: 75
12
63
Enter the operator: *
Enter two numbers: 12   3
36
Enter the operator: /
Enter two numbers: 70    5
14
Enter the operator: x
```

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
    }
}
```

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

name of method

```
public class Main {
    access_modifier return_type method() {
        // code
        return statement;
    }
}
```

f" 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

name → @
naam → @

Creating copy of value of name

i.e., passing value of the reference.

eg 2:

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

Creating copy

name → @
naam → @

name → @
naam → b

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

not changing original object, just creating new object.

✱ 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:

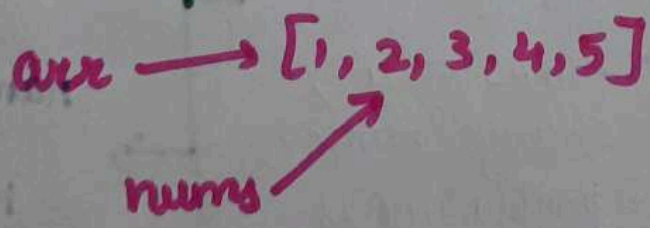
```
psvm() {
    a = 10;
    b = 20;
    swap(a, b);
}
swap(num1, num2) {
    temp = num1;
    num1 = num2;
    num2 = temp;
}
```

a → 10
b → 20
] but not here

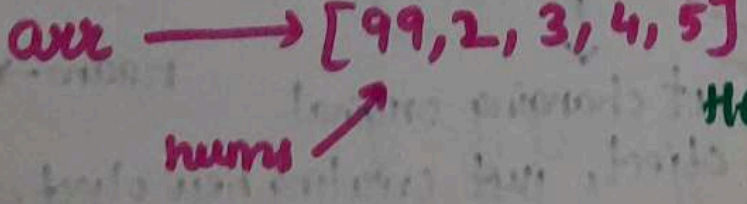
temp → 10
num1 → 20
num2 → 10
] at fn scope level they are swapped.

Here, they just passes the value....

eg-2:



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



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, int b ) {
    //code
}
```
This is allowed having different arguments with same method name.

⇒ At compile time, it decides which fn to run.

⇒ Armstrong number:

Suppose there is number → 153

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

$= 153$

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.

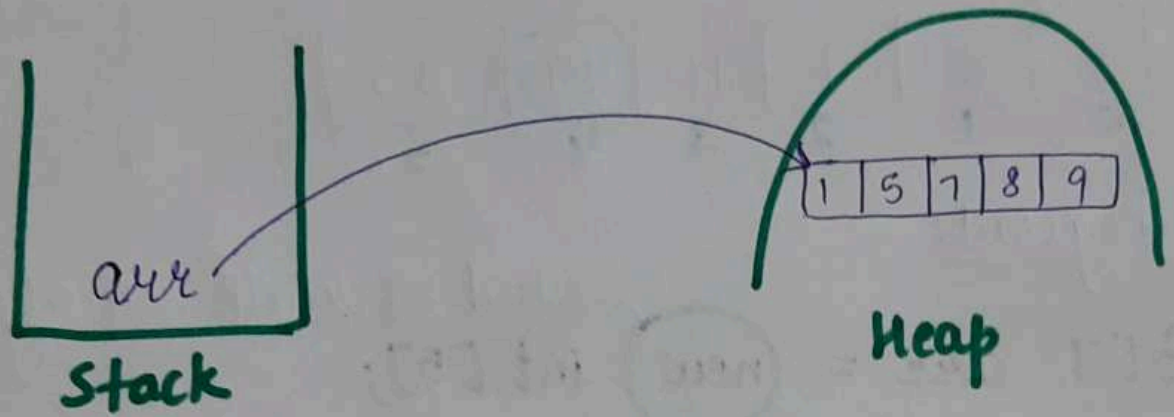
declaration of array → compile time

initialisation → run time

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

- `int []` : datatype
- `arr` : ref var
- `new int[5]` : creating object in heap memory.

⇒ This above concept is known as Dynamic memory allocation which means at runtime OR execution time memory is allocated.



⇒ Internal Representation of Array :

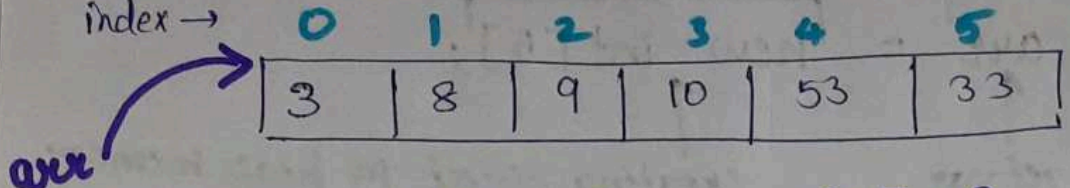
- Internally in Java, memory allocation totally depends on JVM whether it be continuous or not !

Reason 1 : Objects are stored in heap memory.

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

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

⇒ Index of an array:

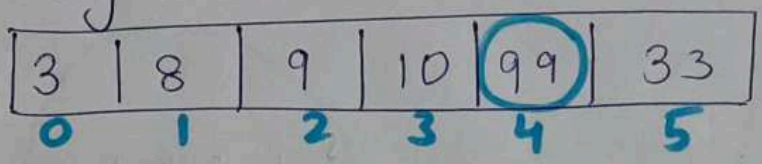


arr[0] = 3
arr[1] = 8
arr[2] = 9
arr[3] = 10
arr[4] = 53
arr[5] = 33

Suppose to change the value of certain index:

arr[4] = 99

New array will be:



⇒ new keyword:

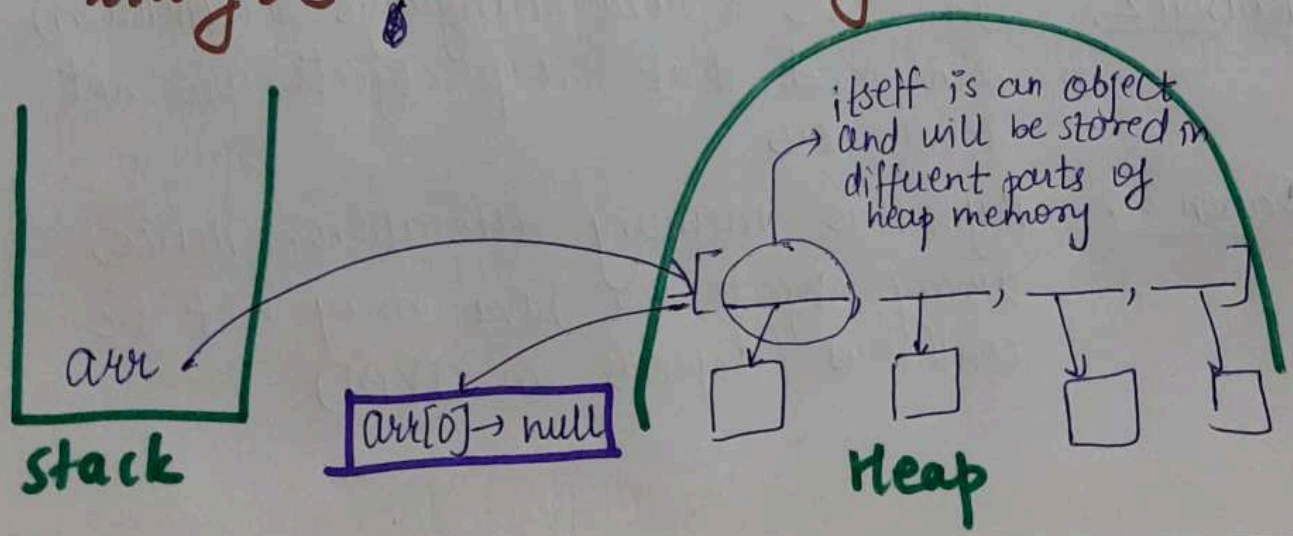
int[] arr = new int[5];

new → used to create an object

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

⇒ If we don't 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];



* 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>

3 (columns)

1	2	3
4	5	6
7	8	9

3 (rows)

⟹ `int[][] arr = new int[size][];`

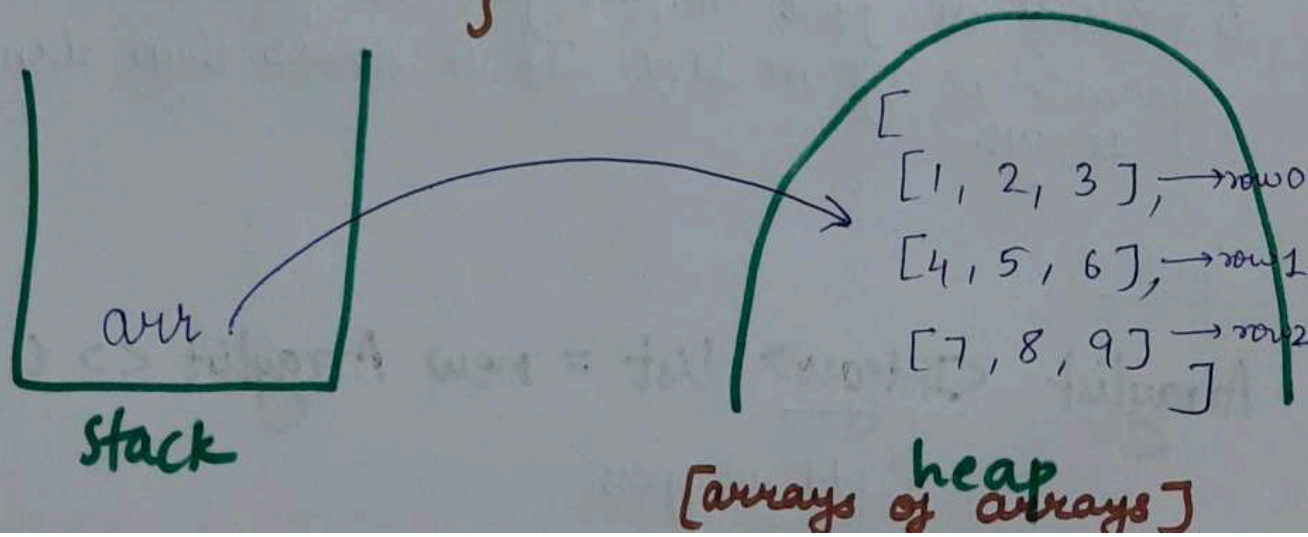
row ↓ column ↓

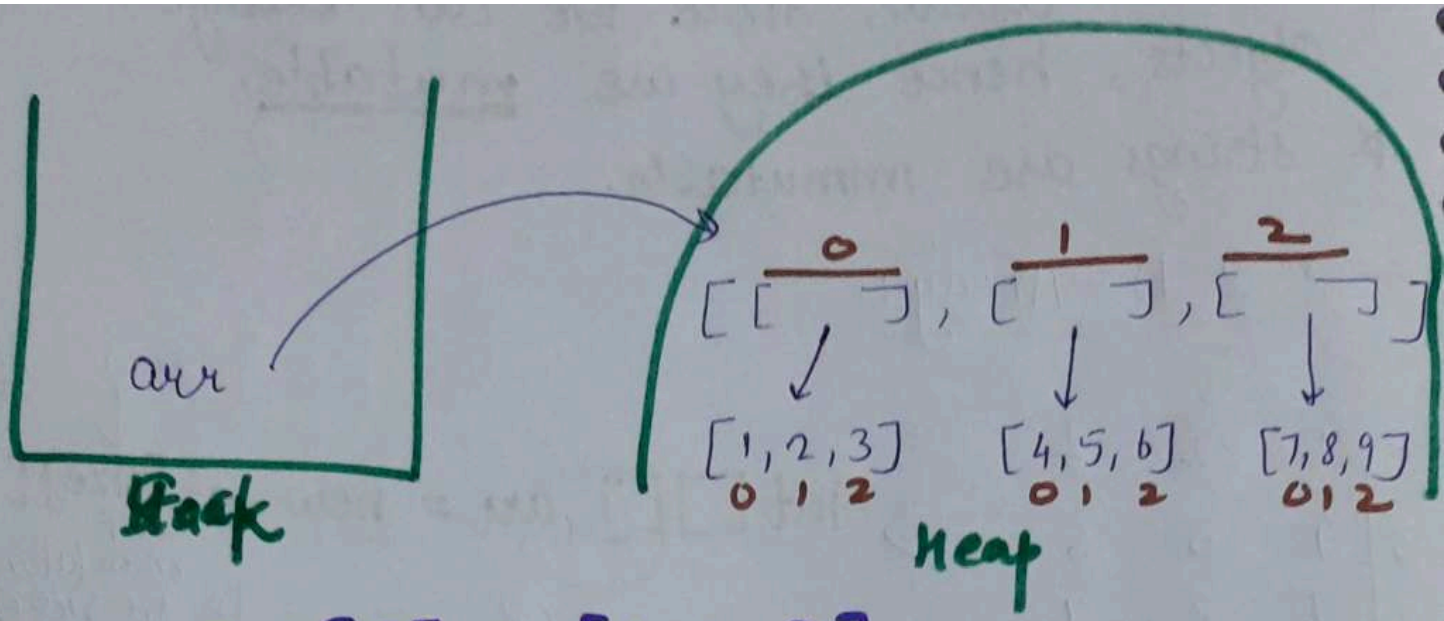
size → mandatory to give size of row

[] → not mandatory

OR

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





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.